UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERA TÉCNICA DE TELECOMUNICACIÓN
ESPECIALIDAD TELEMÁTICA

PROYECTO FIN DE CARRERA

Integración en OMNeT++ de un estimador eficiente de medias como mecanismo de control del fin de la simulación

Autor: Iván de Gracia Moreno
Tutor: Carlos García Rubio

6 de julio de 2009

Agradecimientos

Llegado este momento, no podía olvidarme de dar las gracias a las personas que me acompañaron en este largo viaje por la universidad y que me han permitido, entre otras cosas, que haya podido realizar este proyecto fin de carrera.

En primer lugar, quiero agradecer este proyecto fin de carrera a mi tutor Carlos por sus consejos durante la realización de este trabajo.

A mis compañeros de la universidad, a los de la ingeniería técnica como a los superiores. No tengo suficiente espacio ni tiempo para nombraros a todos, pero gracias por apoyarme en los momentos difíciles y por regalarme la mejor de vuestras sonrisas y vuestra desinteresada ayuda cada vez que lo he necesitado.

En el ámbito personal, quiero agradecer y dedicar este proyecto a mi familia, a mis padres por haber tenido tanta paciencia conmigo todos estos años y haberme apoyado en todo momento sin dejar que me rindiera jamás. A mi hermana Marta, a mis abuelos, a mis tíos y primos por haberme dado tanto cariño.

A mis amigos de toda la vida, que más que amigos son como mis hermanos. Gracias por todos estos años de sincera amistad.

Gracias a todos.

Resumen

La simulación por eventos discretos es una herramienta fundamental para el diseño y desarrollo de todo tipo de sistemas. Sin embargo, los nuevos sistemas son cada vez más complejos, lo que provoca que los modelos de simulación resultantes sean más costosos de resolver, requiriendo más potencia de computación para su evaluación y provocando largos tiempos de espera para la obtención de los resultados.

Este proyecto intenta ofrecer un mecanismo de control para disminuir los tiempos de ejecución de modelos de simulación por eventos discretos, acotando el estado inicial transitorio y permitiendo analizar los datos de salida cuando el sistema ha alcanzado un régimen permanente, asegurando que dichos datos se encuentran en un intervalo de confianza aceptable para tratarlos cómo válidos.

Se ha empleado el entorno de simulación OMNeT++, un simulador modular, con una arquitectura y código libres y un soporte gráfico bastante potente. Para aumentar la calidad estadística de los datos de salida y ahorrar tiempo recogiendo información que no influya en los resultados finales, se ha implementado un criterio de parada, llamado *batch means*, que produce una cobertura estadística superior a otros criterios ya contemplados en el simulador utilizado.

Abstract

The simulation by discrete events is a fundamental tool for the design and development of all type of systems. Nevertheless, the new systems are more and more complex, which causes that the resulting models of simulation are more expensive to solve, requiring more computer power for its evaluation and causing long times of delay of the results.

This project offers a control mechanism to decrease the run times of models of simulation by discrete events, limiting the initial transient state and letting analyze the output data when the system has reached the steady state, ensuring that these data are in an acceptable confidence level to take them as valid.

The OMNeT++ simulation have been used, a modular simulator, with a free architecture and code and a quite powerful graphical support. In order to increase the quality statistical of the output data and to save time collecting information which don't influence on final results, a stopping criterion has been developed, called batch means, that produces a statistical cover superior to other criteria already covered in the simulator tool.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

En muchos estudios de simulación, la mayor parte del coste temporal y económico se pierde en definir el modelo de desarrollo y su programación en la herramienta utilizada para realizar dicha simulación, pero el hecho de analizar apropiadamente la salida de la simulación supone, a priori, un menor esfuerzo. En realidad, el modo de operación más habitual es hacer una única simulación de una longitud un tanto aleatoria y después tratar de estimar el resultado de la simulación con las características de un modelo verdadero.

Históricamente, hay varias razones por las que los análisis de los datos de salida de las simulaciones no se han enfocado de una manera apropiada. En primer lugar, los usuarios a menudo, han tenido la desafortunada impresión, que la simulación sólo es un trabajo de programación en un ordenador, aunque les resulte complicado. En consecuencia, muchos estudios de simulación comienzan por construir un modelo conceptual y su posterior programación, y acaban con una simple ejecución del modelo programado que produce unos determinados resultados. Entonces, si los resultados del estudio de la simulación tienen algún significado, se usan técnicas estadísticas para diseñar y analizar los experimentos de la simulación. Este es uno de los errores más comunes que se producen al estudiar una simulación.

En primer lugar, se deberían tener bien definidos los objetivos al comienzo del estudio y elegir un nivel de detalle apropiado. No debemos tratar un estudio de simulación como si fuera principalmente un problema de programación. Nunca podemos analizar los datos de salida a partir de una sola ejecución, tratándola como la solución verdadera habiendo modelado incorrectamente las distintas fuentes de aleatoriedad como entrada del sistema real.

Un impedimento para obtener una estimación precisa de los parámetros reales del modelo es el coste del tiempo de ejecución para obtener la cantidad necesaria de datos de salida de la simulación. Hay situaciones donde un procedimiento estadístico apropiado es aplicable, pero el coste de tomar la cantidad de datos dictada por el procedimiento es absolutamente prohibitivo. Por fortuna, esta dificultad cada vez se hace menos severa gracias a la mayor velocidad de cálculo de los ordenadores.

Una segunda razón del inadecuado análisis estadístico, es que las salidas de los procesos de todas las simulaciones son no estacionarias y autocorreladas. Por ello, las técnicas estadísticas clásicas basadas en observaciones independientes e idénticamente distribuidas (i.d.d.) no son directamente aplicables.

1.1. Motivación del proyecto

El desarrollo de simuladores ha permitido el estudio de sistemas reales que no se pueden evaluar analíticamente. En consecuencia, esto nos permite utilizar la simulación para validar un modelo analítico. Además con la mejora de los simuladores el ahorro de tiempo se hace notable. Se pueden comparar distintas alternativas de diseño (o de formas de operar de un sistema), para ver cuál se comporta mejor. El ahorro de tiempo es significativo pues hace posible estimar el comportamiento de un sistema existente si se modifican algunas de las condiciones de funcionamiento actuales y, también permite estudiar en poco tiempo, la evolución de un sistema en un periodo largo de tiempo y viceversa.

Sin embargo la simulación no deja de tener algunos inconvenientes. El más obvio, es que la simulación no produce resultados exactos, sino estimaciones. Esto hace necesario el uso de técnicas estadísticas más o menos complejas. Como venimos comentando, desarrollar un modelo de simulación suele ser caro y lleva tiempo. Es difícil demostrar la validez del modelo. Si el modelo no es válido, los resultados son poco útiles, por tanto hay que rediseñar el modelo. Por consiguiente, encontrar el modelo óptimo es difícil: sólo se puede encontrar el mejor entre varias alternativas.

A priori, en una simulación no tenemos por qué saber cuánto tiempo debemos estar recogiendo los datos del problema que estamos estudiando. Si

terminamos demasiado pronto, obtenemos resultados con una variabilidad dependiente de las semillas utilizadas para generar los valores aleatorios. Si terminamos demasiado tarde, estamos desperdiciando tiempo y recursos. Por tanto, lo que nos interesa es el comportamiento del sistema cuando ha alcanzado un estado estacionario y a partir de ahí construir un intervalo de confianza deseado.

Los objetivos tienen un impacto en cómo se debe ejecutar la simulación. Por ejemplo, si el objetivo es comprender el comportamiento del sistema, entonces se podría desear una salida de presentación agradable y velocidad de simulación controlable con condiciones y parámetros iniciales ajustables. Si la simulación se hace para hacer estimaciones de variables de salida, entonces se deben proveer intervalos de confianza de determinada precisión, y la simulación se debe ejecutar tan rápido como sea posible. En la selección de parámetros, la simulación se puede ver como parte de un bucle de optimización. La información obtenida por la simulación puede consistir en estimaciones aleatorias correspondientes a los parámetros dados y ser usados en algún método de aproximación estocástica, o consistir también en gradientes de estimaciones y ser usados en algoritmos de gradientes estocásticos.

Las propiedades estadísticas han de ser comparadas antes de seleccionar la metodología apropiada. En el caso del ajuste de un modelo, se necesitará ejecutar una cantidad de simulaciones con diferentes parámetros y las variables de salida de interés y sus propiedades estadísticas también necesitarán ser investigadas.

Los eventos que ocurren en el sistema modelado pueden tener su origen en una traza tomada de un sistema real o pueden generarse de forma aleatoria. En el segundo caso, estamos ante un problema que necesita de herramientas estadísticas para su análisis, siendo un ingrediente fundamental un buen generador de números pseudoaleatorios para la generación aleatoria de los eventos. Suponiendo que el modelo de simulación por eventos discretos es alimentado con un generador de números aleatorios, la salida de dicho modelo será una respuesta a la entrada, produciéndose muestras estadísticas de una población desconocida.

En el presente proyecto lo que nos interesa estimar es el valor medio de una muestra $E[X]$ (o esperanza matemática) de una variable aleatoria de salida del modelo, es decir, hacer lo que se denomina una estimación puntual. Sin embargo, la única manera de obtener dicha esperanza es teniendo una muestra de infinitos valores, lo cual no es posible con un tiempo limitado

de computación. Como máximo se podrá ofrecer una estimación $\hat{X}$ de la verdadera esperanza $E[X]$ dentro de un intervalo de confianza H, es decir, dentro de un intervalo $(X - H, X + H)$ donde hay una determinada probabilidad, normalmente alta, de que $E[X]$ esté dentro de ese rango.

Si nos fijamos en la literatura, en el texto de Law y Kelton [4], la estimación de la esperanza y de los intervalos de confianza no es un problema estadístico totalmente resuelto. Hay diferentes maneras de resolverlo, cada una con sus ventajas e inconvenientes, pero siempre se tendrá una mayor o menor duda sobre la validez del resultado.

Este es un problema común encontrado por los simuladores: el construir un intervalo de confianza (i.c.) para la media μ. Existen dos aproximaciones básicas para ello: construir un i.c. a partir de un tamaño de muestras arbitrario fijo o incrementar secuencialmente el tamaño de muestras hasta construir un i.c. aceptable, momento en que se detendrá la simulación.

Para el segundo caso, que es el de nuestro interés, existen al menos cinco métodos diferentes que podrían ser aplicados. Uno de ellos es el método de media de bloques o *batch means* que más adelante detallaremos.

1.2. Objetivos

El objetivo inicial de este proyecto es encontrar un mecanismo para controlar el final de una simulación e integrarlo dentro del simulador OMNeT++.

Como hemos expuesto anteriormente, en muchas simulaciones el tiempo de observación de los datos se desconoce, con lo que no podemos asegurar que los datos de salida si detenemos la simulación en un cierto instante, sean lo suficiente independientes unos de otros para tomar dichos datos como aceptables. Esto provoca la necesidad de encontrar un mecanismo capaz de analizar los datos de la simulación y estimar si estos están lo suficientemente no relacionados para asegurar una cierta independencia y aceptarlos como válidos, en lugar de, realizar muchas ejecuciones de la simulación para obtener mayor independencia entre los datos.

El método que hemos implementado, *batch means*, subdivide una simulación muy extensa temporalmente, en bloques separados que juegan el papel de repetir varias ejecuciones de una misma simulación. Este método calcula un estimador del parámetro que estemos estudiando para decidir si,

en función de la cobertura de aproximación al valor real, podemos dar por finalizada la simulación.

Hay que reseñar, que el simulador OMNeT++ dispone de un mecanismo para llevar a cabo el cometido que nos planteamos. Sin embargo, es un método que debido a su sencillez matemática (que más adelante se discutirá), no ofrece unos resultados tan satisfactorios como el método *batch means*.

1.3. Contenido de la memoria

El contenido de esta memoria se encuentra dividido en los siguientes capítulos:

- Capítulo 1: en este capítulo, se presenta la motivación por la cual ha surgido este proyecto, cuál es su origen así como los objetivos planteados en el desarrollo del mismo, la importancia y utilidad de encontrar un mecanismo para controlar el final de una simulación y la obtención de unos resultados ajustados y fiables.

- Capítulo 2: en este capítulo, se explican diferentes métodos de análisis de datos de una simulación, entre ellos el de nuestro objeto de estudio y además se hace una introducción sobre los mecanismos de los que dispone la herramienta de simulación OMNeT++.

- Capítulo 3: en este capítulo, se indican cuáles han sido las decisiones de implementación para integrar la librería dentro de la herramienta de simulación OMNeT++.

- Capítulo 4: se muestran los resultados obtenidos por el algoritmo de *batch means*, al tratar de simular un sistema de colas M/M/1 y M/M/2, y un escenario con 20 hosts mandando paquetes aleatoriamente hacia un servidor utilizando un medio de acceso compartido y usando el protocolo de acceso Aloha. Se incluyen los comentarios más relevantes obtenidos mediante la realización de estas pruebas, así como conclusiones prácticas obtenidas de las mismas.

- Capítulo 5: en este capítulo, se muestra un resumen de las etapas que se han llevado a cabo para realizar este proyecto.

- Capítulo 6: se realizan algunas conclusiones acerca del diseño realizado y otras posibles vías para resolver el problema de encontrar un mecanismo eficaz capaz de delimitar el final de una simulación.

También se incluyen en la memoria una serie de anexos para que el lector de esta memoria pueda encontrar rápidamente una ayuda que le permita conocer, por ejemplo, el código de la librería implementada, así como guías de uso de la herramienta de simulación OMNeT++.

Para concluir, se incluye un apartado con toda la bibliografía utilizada a lo largo del desarrollo de este proyecto.

Capítulo 2

Estado del arte

2.1. Estados de la simulación

Anteriormente hemos analizado cuáles son los dos principales problemas que encontramos al realizar una simulación sin una duración determinada: cuándo es correcto usar las muestras producidas por la simulación y cómo detectar que hemos recogido las suficientes muestras que satisfagan unos requerimientos de precisión. Por ejemplo, en un sistema de colas, los tiempos de espera en cola son pequeños al principio, pero después de cierto tiempo, el sistema se estabiliza, y los tiempos empiezan a ser similares. Se dice entonces que el sistema ha alcanzado el estado permanente o estacionario.

Si la simulación se inicia en un estado específico (por ejemplo, el sistema vacío y el servidor desocupado, en una cola con un único servidor), y se ejecuta hasta que se produce un suceso determinado que identifica el fin de la simulación (por ejemplo el fin de la jornada de trabajo) hablamos de una simulación con horizonte finito. En este caso, el sistema puede no alcanzar el estado estacionario y la estimación de cualquier parámetro a partir de los datos de salida del sistema dependerá del transitorio[1], y por tanto de las condiciones iniciales, lo que supone un gran inconveniente, ya que solamente cuando la fase transitoria ha acabado los datos pueden ser recogidos. Existen muchas técnicas diferentes que analizan cuándo finaliza este estado.

Todas las técnicas más complejas asumen que la varianza de las muestras es menor en el estado permanente que en la fase inicial. Esta suposición

[1]Las funciones de distribución que caracterizan las muestras i, al inicio ($i \to 0$) serán diferentes, pero las distribuciones convergerán a un función de distribución común en el estado permanente a medida que i aumenta ($i \to \infty$). El rango de i donde las distribuciones no han convergido todavía se denomina estado transitorio

es habitual pero no necesariamente verdadera. Tomar sólo una ejecución implica pasar una sola vez por el estado transitorio, lo que nos otorga una mayor eficiencia. Si recordamos, la media muestral es por definición un estimador insesgado de la media poblacional μ, pero como las X_i son, en general, variables aleatorias dependientes (por ejemplo, en muchos sistemas de colas están correlacionadas positivamente), el estimador de la varianza está fuertemente sesgado. Los estimadores, como ya hemos dicho, son funciones usadas para determinar un parámetro desconocido de una distribución de población basándose en las muestras observadas. Por ello en este tipo de simulaciones lo que se hace es ejecutar k repeticiones independientes de la simulación del sistema. Cada repetición parte del mismo estado y utiliza muestras independientes de números pseudoaleatorios. De esa forma se consigue tener k medias muestrales independientes y poder calcular un estimador insesgado de la varianza muestral.

En el caso de los modelos de colas se puede demostrar (ver [4]) que el uso de todas las observaciones minimiza el error cuadrático medio:

$$Error = E[X - \mu]^2 = VAR(X) + [E[X] - \mu]^2 \tag{2.1}$$

Por definición, como se demuestra en los libros (ver [6]), un estimador insesgado cumple que $E[X] = \mu$, y entonces $Error = VAR(X)$. Sin embargo, se puede demostrar que un error cuadrático mínimo no garantiza que se alcance adecuadamente un nivel de precisión α. La media muestral también es un estimador consistente para la media:

$$\lim_{n\to\infty} P(|1/n\sum_{i=1}^{n} x_i - \mu| < \varepsilon) = 1 \;\; \forall\varepsilon > 0, \forall\mu \tag{2.2}$$

siendo n el tamaño de la muestra.

En general, las variables X_i observadas están correlacionadas positivamente (por ejemplo, las longitudes de las colas o los tiempos medios de espera en una cola), y constituyen un proceso estacionario en sentido amplio, es decir, sus funciones media y varianza son constantes (e iguales a μ y σ^2 respectivamente) y su función de covarianza satisface: $cov[X_i, X_{i+k}] = C(k)$, es decir depende únicamente de la distancia y en consecuencia la función de autocorrelación está definida por:

$$\rho(k) = \frac{cov[X_i, X_{i+k}]}{\sigma^2} = \frac{C(k)}{\sigma^2} \tag{2.3}$$

Sin embargo, cuando la simulación no tiene un horizonte finito, se supone que la estimación de interés corresponde al estado estacionario del sistema simulado, para lo cual hay que eliminar la fase transitoria.

2.1.1. Definición del estado permanente

Definimos $F_i(y|I) = P(Y_i < y|I)$ como la función de distribución de Y_i dadas las condiciones iniciales I, siendo dicha función en general distinta para cada valor de i y para cada condición inicial dada I. Si existe una función de distribución $F(y)$ tal que:

$$F_i(y|I) \rightarrow F(y) \; si \; i \rightarrow \infty, \forall I, y \tag{2.4}$$

entones a F(y) se le llama distribución del estado permanente del proceso Y_1,Y_2, ..., Y_i. Hay que tener en cuenta que F(y) no existe siempre necesariamente.

La figura 2.1 que se muestra a continuación, muestra que las funciones de densidad de los procesos estocásticos a lo largo de la simulación tienen una media muestral en régimen permamente mucho más parecida que al inicio de la simulación durante la fase transistoria.

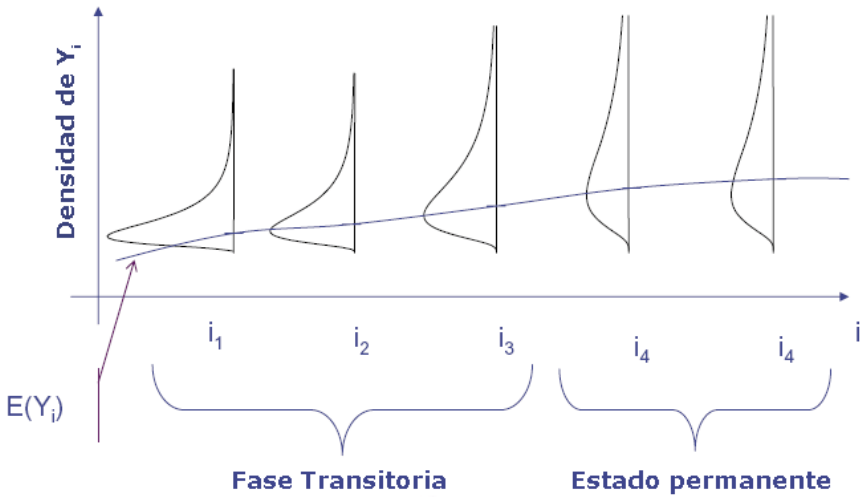

Figura 2.1: Figura tomada de Simulation Modeling and Analysis de Law A. M. y Kelton W. D.

Por ejemplo, si estudiamos un sistema de colas, una única ejecución de la simulación producirá una estimación del tiempo medio de espera en cola. Pero es sólo una simple muestra de un proceso estocástico, y esto resulta insuficiente. Múltiples ejecuciones producirán una muestra cada vez. Si las repetidas ejecuciones se hacen usando los mismo valores de parámetros, a pesar de generar diferentes números aleatorios, los resultados podrían ser comparables. Esto es, las muestras repetidas de los tiempos medios de espera son muestras de variables aleatorias independientes e idénticamente distribuidas. Sin embargo, esta distribución de los tiempos medios de espera es desconocida.

2.1.2. Métodos para el análisis de datos

En el caso que interesa, la salida de una ejecución del modelo de simulación genera observaciones $X_1, X_2, \ldots, X_N$ que generalmente son muestras de una serie temporal autocorrelada y cuya medida de rendimiento μ se estima en régimen permanente como,

$$\bar{\mu} = \lim_{N \to \infty} \frac{1}{N} \sum X_i \tag{2.5}$$

En el proceso de simulación de régimen permanente hay que preguntarse, tarde o temprano, cómo de extensa en el tiempo ha de ser para conseguir una determinada precisión. Es una cuestión difícil de responder que a menudo se tiende a ignorar. Muchas veces se hacen las simulaciones tan largas como dicte la intuición o la paciencia del usuario, y los valores obtenidos con una sola ejecución son tomados como la verdadera solución, sin tener en cuenta su precisión.

El propósito de este apartado es revisar las consideraciones y las aproximaciones para estimar y controlar la precisión de la salida, pues uno o varios de estos métodos deberán emplearse en este trabajo para realizar el análisis de la salida de los modelos.

Los métodos más usados (ver [4] pág 528-529) para estimar el intervalo de confianza para la media de una variable de salida de una simulación son: réplicas, lotes, regeneración, autorregresión, análisis espectral y series temporales estandarizadas. Los tres primeros métodos intentan eliminar los efectos de la correlación, mientras que el cuarto y el quinto intentan estimarla; el último método calcula el intervalo de confianza basándose en propiedades de una transformación de la secuencia de salida de la simulación.

Las implementaciones para la estimación del intervalo de confianza se pueden clasificar en procedimientos con tamaño de muestra fija o procedimientos secuenciales. La mayoría de los métodos de análisis se pueden adaptar a cualquiera de los procedimientos. En un procedimiento de tamaño de muestra fija, se realiza una simulación de longitud total prefijada, y el intervalo de confianza se estima de los resultados una vez completado el experimento, lo cual no permite controlarlo. En un procedimiento secuencial, se especifica la precisión deseada, y el experimento continua hasta que se obtiene esta precisión.

A continuación se muestra una tabla que resume la problemática de cada uno de los métodos de análisis de datos en la salida descritos anteriormente:

Método	**Número de réplicas**	**Problema de sesgo principal**	**Mayor dificultad**
Réplicas-borrado	n (n$\geq$ 2)	$\bar{X}(n)$	Elección del período de calentamiento
Batch Means	1	$\hat{Var}[\bar{X}(n)]$	Elección del tamaño de lote para obtener lotes sin correlación
Autorregresivo	1	$\hat{Var}[\bar{X}(n)]$	Calidad del modelo autorregresivo
Espectral	1	$\hat{Var}[\bar{X}(n)]$	Elección del número de saltos de covarianza
Regeneración	1	$\hat{Var}[\bar{X}(n)]$	Existencia de ciclos con tamaño de la media
Series temporales estandarizadas	1	$\hat{Var}[\bar{X}(n)]$	Elección del tamaño del lote

Cuadro 2.1: Métodos para el análisis de datos de la simulación

2.1.3. El método Batch Means

Introducción

El método de media de bloques, que pasamos a describir a continuación, se basa en la suposición de que las observaciones más separadas en el tiempo están menos correladas. De esta forma, para bloques de observaciones suficientemente largos, las medias de los bloques estarán prácticamente incorreladas. Un determinado tamaño de bloque es aceptado como el correcto si todos los coeficientes de autocorrelación son estadísticamente despreciables para un nivel significativo, β_k, $0 < \beta_k < 1$.

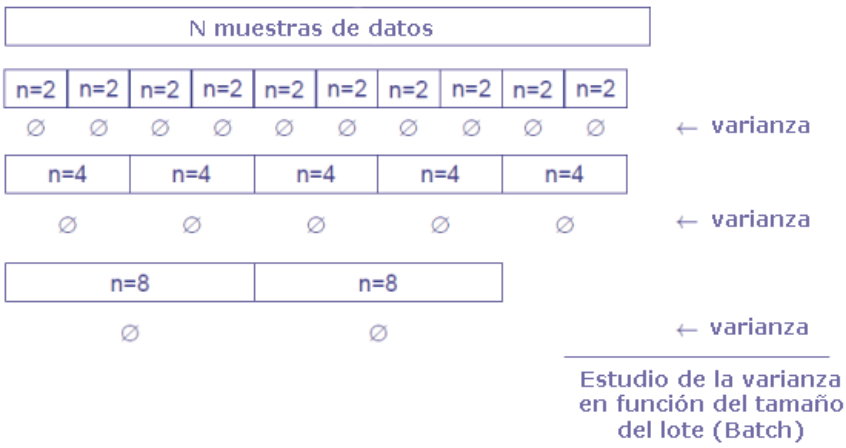

Figura 2.2: Mecanismo de Batch Means. Figura tomada del artículo Leistungsbewertung and Simulation de Holger K.

Para cada bloque separado se calcula su media. Las decisiones son tomadas estudiando la varianza de esta secuencia de medias de bloques cuando el tamaño de los bloques varía. Asumimos la longitud del período transitorio como T. Los bloques del estado estacionario tienen todos la misma media a diferencia de las medias del estado transitorio. Por lo tanto, muchas medias de bloques diferentes, dan como resultado una gran varianza en el conjunto de las medias de bloques. Cuando el tamaño del bloque crece, cada vez menos y menos bloques pertenecerán al transitorio. Cuando el tamaño del bloque excede T, sólo un bloque será del periodo transitorio. Estos pocos bloques tendrán un mínimo impacto en la varianza. Eventualmente, sólo una media de bloque será diferente, y la varianza será pequeña comparada con el caso de los bloques con tamaño menor.

La finalidad del método es realizar la división de una ejecución larga en un conjunto de k en subejecuciones de longitud m, llamadas lotes, (del inglés *batch*), calculando una media de la muestra para cada lote, y utilizándola para calcular una media global y el intervalo de confianza. Como ventaja adicional, este método reduce el problema del transitorio inicial al sólo necesitar eliminar uno.

Parámetros característicos

Supongamos que hacemos una simulación de longitud n observaciones y dividimos los datos resultantes $X_1, X_2, \ldots, X_n$ que asumimos forman parte de un proceso estacionario covariante, en k bloques de m observaciones, siendo $n = k \cdot m$. Las medias de las n observaciones en cada uno de estos bloques m, $\bar{X}_1(m), \bar{X}_2(m), \ldots, \bar{X}_k(m)$, son usados como datos de salida secundarios para el análisis de estadístico de los resultados.

Según lo dispuesto en el anterior párrafo, la media muestral total es:

$$\bar{\bar{X}} = \frac{1}{k} \sum_{j=1}^{k} \bar{X}_j \tag{2.6}$$

Si estimamos la varianza de dicha media, y escogemos m lo suficientemente grande para que las medias $\bar{X}_j(m)$ sigan aproximadamente una distribución normal además de estar incorreladas, podemos tratarlas como si $\bar{X}_j(m)$ fueran variables independientes e idénticamente normalmente distribuidas, y podemos construir un intervalo de confianza para un nivel de confianza $100(1-\alpha)\,\%$, siendo α la precisión deseada, como

$$\bar{\bar{X}}(k,m) \pm t_{k-1,1-\alpha/2} \sqrt{\hat{\sigma}^2[\bar{\bar{X}}(k,m)]} \tag{2.7}$$

donde $t_{k-1,1-\alpha/2}$ es el punto crítico superior $(1 - \frac{\alpha}{2})$ obtenido de la función t de Student con *k-1* grados de libertad.

Los coeficientes de autocorrelación los estimamos mediante los llamados estimadores *jackknife*, que suelen ser los menos sesgados (ver Law y Carson [3]). Los estimadores *jackknife* del coeficiente de autocorrelación de retraso k para una secuencia de medias de bloques de tamaño m se calculan a partir de la siguiente expresión:

$$\tilde{\rho}(k,m) = 2\hat{\rho}(k,m) - \frac{\hat{\rho}'(k/2,m) + \hat{\rho}''(k/2,m)}{2} \tag{2.8}$$

donde los estimadores de la derecha se calculan como coeficientes de autocorrelación ordinarios, exceptuando que $\hat{\rho}(k,m)$ se estima sobre las k medias de bloques, mientras que $\hat{\rho}'(k,m)$ y $\hat{\rho}''(k,m)$ se estiman, respectivamente, sobre la primera y la segunda mitad del bloque analizado (asumiendo que k es par) y donde dichos estimadores se calculan según la expresión:

$$\hat{\rho}_1(k,m) = \frac{\sum_{j=1}^{k-1}\left[\bar{X}_j - \bar{\bar{X}}(k,m)\right]\left[\bar{X}_{j+1} - \bar{\bar{X}}(k,m)\right]}{\sum_{j=1}^{k}\left[\bar{X}_j - \bar{\bar{X}}(k,m)\right]^2} \tag{2.9}$$

Posibles inconvenientes

La forma de trabajo descrita en el párrafo anterior tiene un inconveniente; valores pequeños de k tienden a aumentar la varianza, por lo que será deseable un método que permitiese aumentar k sin disminuir m para una cantidad de muestras fija. Una idea para resolver esta problemática es el método denominado lotes solapados (del inglés *overlapping batchs*) que puede leerse en el artículo publicado por Meketon y Schemiser (ver [5]). En este caso se utilizan $n - m + 1$ lotes para obtener el estimador

$$s^2 = \frac{1}{n-m+1}\sum_{i=1}^{n-m+1}\frac{(X_j - \bar{X})^2}{n/m - 1} \tag{2.10}$$

con X_j calculado a partir del lote de tamaño m que va de x_j a x_{j+m-1}. Hay soluciones intermedias con estimadores de menor solapamiento que el anterior.

Existe una posibilidad adicional que consiste en eliminar un conjunto de datos de salida entre cada lote, lo que, en principio, disminuye el peligro de correlación entre lotes a costa de perder muestras. Además, se demuestra que para tamaños de bloque suficientemente grandes, las medias de bloque están distribuidas normalmente.

Esto es cierto a causa de versiones extendidas del teorema central del límite que puede considerarse cierto incluso para algunos tipos de procesos correlados. Debemos evitar el problema de la correlación usando las media de un subconjunto de datos para calcular los intervalos de confianza. Si los subconjuntos están apropiadamente escogidos, las medias están incorreladas. Las formas habituales para calcular intervalos de confianza pueden ser usadas entonces. Sólo estamos realizando una ejecución de la simulación. Esto nos proporciona una ventaja y es que sólo pasamos una vez por el

estado transitorio.

Asumiendo que se emplea el borrado para conseguir alcanzar las condiciones para el estado estable, cada lote empezará con el sistema en este estado. Si el tamaño del lote es suficientemente grande, las medias serán aproximadamente independientes y distribuidas normalmente. Se puede calcular un intervalo de confianza igual que para las réplicas, con las medias de lote tomando el lugar de las medias de las ejecuciones, es decir, donde X_i es la media de las m muestras de cada lote.

Como se ha dicho, el efecto de la dependencia dentro de cada lote implica sesgos en X_i y, por lo tanto, también en $\bar{X}$, que se reduce al aumentar m, por lo que si m es suficientemente grande, la dependencia no tiene prácticamente efecto dentro del lote. Aunque se espera cierta dependencia entre los estimadores de lotes, ésta será menor que la dependencia entre observaciones.

El efecto del arranque sólo se debe de tener en cuenta una vez, mientras que en el caso de las réplicas debe ser k veces. El método por lotes es potencialmente más eficiente, pues se necesita un menor número de muestras para conseguir una precisión dada. El mayor problema consiste en determinar el tamaño adecuado del batch para garantizar que cada lote es independiente. Desgraciadamente no hay un método sencillo para elegir un tamaño del lote o el número de lotes.

Podemos encontrar algún autor como Andreas Willig (ver [17]) que sugiere encontrar el tamaño del lote probando diferentes valores de m, esto es, $m_1 < m_2 < ... < m_i$ de tal modo que siempre sean potencias de 2, y calcular la media de cada bloque de m_i/m_1 hasta encontrar un valor donde la media éste acotado significativamente entre dos valores relativamente pequeños.

También podemos encontrar autores que estiman el tamaño de los lotes a partir de la verificación de la independencia de las medias de los lotes por medio del estadístico de Von Neumann:

$$\frac{\sum_{i=2}^{m}\left(X_i - X_{i-1}\right)^2}{\sum_{i=1}^{m}\left(X_i - \bar{X}_k\right)^2} \approx N\left(2, \frac{4(k-2)}{(k^2-1)}\right) \tag{2.11}$$

2.2. El simulador OMNeT++

2.2.1. Introducción

La historia del desarrollo de Objective Modular Network Testbed in C++ (OMNeT++)[2] no es reciente. Su autor, András Varga, empezó a desarrollar este sistema en 1992 a partir del simulador Omnet escrito en Pascal desarrollado a su vez por su profesor en la Universidad Técnica de Budapest, el Dr. Gyrgy Pongor.

Actualmente son muchas las universidades que están apostando por el uso y desarrollo de esta herramienta ya que la licencia académica es pública. También las grandes empresas de telecomunicaciones como Cisco, Alcatel-Lucent, Orange, IBM, Intel o HP han adquirido licencias comerciales (OMNEST) para su uso.

OMNeT++ es un simulador de eventos discretos modular, orientado a objetos. Un modelo en OMNeT++ consiste en módulos jerárquicamente anidados, que se comunican mediante paso de mensajes. Presenta dos interfaces de ejecución. Una gráfica y otra de comandos. El modo gráfico permite una simulación paso a paso (de mensajes), o de forma continua (con diferente granularidad). La interfaz visual, además, es una herramienta didáctica y de depuración fundamental. Existe un gran esfuerzo desarrollador, tanto del entorno de simulación como de las librerías (IPv6, TCP, Mobility ...). Con el entorno de OMNeT++ se pueden ejecutar simulaciones compiladas que requieran la herramienta gráfica, mientras que para desarrollar nuevas es preciso implementar los módulos y compilarlos. Implementa la orientación a procesos y la orientación a eventos, y fuerza a una jerarquización de los modelos, lo que facilita una programación ordenada así como la reutilización de código.

Para poder llevar a cabo la simulación es preciso compilar primero el código fuente. Por ello, la mayor dificultad consiste en obtener dicho código fuente (bien por librerías disponibles, bien por desarrollo del usuario) y ejecutar un compilador de C++. Además, las herramientas visuales son bastante útiles para depurar el código (frente a otros simuladores como NS-2, esto puede suponer una gran ventaja). Las librerías disponibles se encuentran, por lo general, muy bien documentadas y debido al desarrollo modular de las mismas, es posible emplear únicamente aquellos bloques que

[2]http://www.omnetpp.org

sean de nuestro interés.

Recientemente se publicó la versión 4.0 (marzo 2009) que introduce algunas novedades, entre ellas la integración de todo el interfaz de desarrollo, basado en el entorno Eclipse[3]. Durante el desarrollo del proyecto, se usó inicialmente la versión 3.3 (la última hasta entonces). Para poder usar las clases desarrolladas en la nueva versión se migraron los archivos para que fueran compatibles. Afortunadamente, existen scripts automáticos que realizan esta migración[4].

Cualquier escenario que nos planteemos simular con la herramienta OMNeT++, deberá estar formado por módulos que tendrán un comportamiento particular definido por el programador que se ajuste a las necesidades del modelo a estudiar.

Cada módulo simple se programa como una clase C++ que hereda de la clase cSimpleModule. OMNeT++ utiliza el paso de mensajes para la comunicación entre módulos. Existen dos modelos diferentes de tratamiento de mensajes:

- **Síncrono**
  - Se define un bucle completo de tratamiento
  - Más sencillo de comprender inicialmente
  - Requiere espacio de pila de memoria
- **Asíncrono**
  - Disparado por eventos
  - Más costoso de comprender
  - No requiere espacio de pila de memoria

En nuestro caso, usaremos el modelo asíncrono ya que, aunque es más complejo de implementar, ofrece más garantías y es mucho más sofisticado.

[3]http://www.eclipse.org
[4]Más información en http://omnetpp.org/doc/omnetpp40/migration/migration.html

2.2.2. Mecanismos de detección de OMNeT++

La detección del final del período transitorio y la precisión de resultados son soportadas por OMNeT++ desde sus versiones anteriores a la versión 4.0 (ver sección 6.8.4 del manual de usuario [15]).

El usuario puede referenciar los objetos de detección del transitorio y de precisión de resultado a un objeto resultado de la clase cStatistic y mediante unos algoritmos comprobar si el período transitorio ha acabado o si hemos alcanzado la precisión deseada en el resultado. Las clases que implementan dichos algoritmos son:

- cTransientDetection
- cAccuracyDetection

cuyos principales métodos son:

- addTransientDetection(cTransientDetection *object);
- addAccuracyDetection(cAccuracyDetection *object);
- cTransientDetection *transientDetectionObject();
- cAccuracyDetection *accuracyDetectionObject();

Detección del estado transitorio

Existe una librería que implementa un algoritmo, en la clase cTDExpandingWindows (que hereda de cTransientDetection), que usa una aproximación de ventana deslizante con dos ventanas, y comprueba la diferencia de las dos medias para ver si el periodo transitorio ha finalizado. A través del método setParameters de esta clase podemos ajustar los parámetros del algoritmo como el tamaño de la ventana o el número de repeticiones necesarias para la detección.

```
void setParameters(int reps=3, int minw=4, double wind=1.3, double
acc=0.3);
```

Detección de la precisión

También existe otra librería que implementa un algoritmo, en la clase cADByStddev (que hereda de cAccuracyDetection), que divide la desviación estándar por el cuadrado del número de muestras recogidas y comprueba si el valor obtenido es lo suficientemente pequeño para determinar una cierta precisión en los resultados. El método *setParameters* permite ajustar los parámetros del algoritmo como la precisión o el número de repeticiones que queremos ejecutar para obtener el resultado.

```
void setParameters(double acc=0.1, int reps=3);
```

Como se puede comprobar, la herramienta OMNeT++ dispone de mecanismos para poder tratar y detectar el final de simulación. Sin embargo estos mecanismos son válidos pero algo simples en cuanto a la profundidad y la complejidad del análisis estadístico, por lo que se hace necesario implementar otros mecanismos que resuelvan el problema de la eliminación del estado transitorio en la simulación de un modelo analítico.

Capítulo 3

Desarrollo de la librería e integración con OMNeT++

3.1. Arquitectura de la programación

Como se ha señalado anteriormente, OMNeT++ es un simulador desarrollado en lenguaje C++ y un lenguaje propio llamado NED. El primero se usa para dar la funcionalidad a los objetos o módulos que forman el modelo que vamos a simular, pudiendo usar diferentes librerías ya implementadas, mientras que el lenguaje NED define los módulos que forman parte del modelo (entrada y salida de datos, conexiones con otros módulos, etc.)

Por tanto en el desarrollo del proyecto, por una parte se implementó el algoritmo de *batch means*, para dar lugar a una librería independiente que pueda ser usada en cualquier modelo a simular.

La librería **cADByBatchMeans** implementa un procedimiento (en lenguaje C++) que calcula un estimador de la media de las muestras recogidas en la simulación para definir un mecanismo que permita controlar el final de la simulación.

Posteriormente, para poder demostrar la eficacia del procedimiento implementado, desarrollaremos o utilizaremos algún escenario o modelo del que pretendamos realizar un estudio sobre algún parámetro característico. En el caso de este proyecto, por ejemplo, la media del tiempo de espera en una cola de un paquete o el tráfico cursado en un canal de datos.

3.2. Descripción del algoritmo implementado

El texto de Law y Carson (ver [3] pág 1016), establece el procedimiento a seguir para una correcta implementación del algoritmo, es decir, del método de *batch means*. Básicamente se trata de implementar unas funciones que calculen la correlación de las muestras y calcular un intervalo de confianza que está dentro del rango deseado. Seguidamente, se muestra la traducción a lenguaje C++ de los pasos necesarios para desarrollar el mecanismo a implementar en una librería compatible con el entorno OMNeT++:

- Paso 0. Habiendo ajustado los parámetros iniciales apropiadamente (ver más adelante la explicación de cada uno de ellos), sea $i = 1$, recogemos $n_i = 800$ muestras. Consideramos $l \cdot k = 400$ batches de tamaño $m = 2$.

```
#define CorrelationThreshold 0.4
#define InitialSamples 600
#define StepSamples 200
#define FinalBatches 40

#define BatchesAccumulatorLength 400
```

- Paso 1a. Dividimos las n_i observaciones en lk bloques de tamaño m. Calculamos el estimador de correlación $\tilde{\rho}_1(lk, m)$ a partir de $\bar{X}_j(m)(j = 1, 2, \ldots, lk)$. (ver ecuación 2.9 del anterior capítulo)

```
...

CollectionCounter++;

BinaryBatch = BinaryBatch+Sample;
BinaryBatchIndex++;

if(BinaryBatchIndex == BinaryBatchSamples)
{

   BinaryBatchesAccumulator[BinaryAccumulatorIndex]
     = BinaryBatch/BinaryBatchSamples;

   BinaryBatch = 0.0;
```

```
   BinaryBatchIndex = 0;
   BinaryAccumulatorIndex++;

}

...

for(i=0 to BatchesAccumulatorLength / 2)
{

   BinaryBatchesAccumulator[i] =
     (BinaryBatchesAccumulator[i*2 + 1] + BinaryBatchesAccumulator[i*2])/2.0;
}

BinaryAccumulatorIndex = (BatchesAccumulatorLength / 2);
```

Si $\tilde{\rho}_1(lk, m) \geq$ c , ir al Paso 2.
Si $\tilde{\rho}_1(lk, m) < 0$ ir al Paso 1c. En otro caso ir al Paso 1b.

```
if(CollectionCounter == CollectionSamples)
{
   ...
   FirstCorrelationEstimate =
            JackKnifing(0,BatchesAccumulatorLength);

   ...

   SecondCorrelationEstimate =
            JackKnifing(0,BatchesAccumulatorLength / 2);

   ...

   if(FirstCorrelationEstimate >= CorrelationThreshold)
   {

      go Paso 2

      if(FirstCorrelationEstimate <= 0.0
         || FirstCorrelationEstimate > SecondCorrelationEstimate)
```

```
      {

          go Paso 1c
            ...
      }
   }
}
```

- Paso 1b. Dividir n_i en $lk/2$ bloques de tamaño $2m$. Calcular $\tilde{\rho}_1(lk/2, 2m)$ a partir de $\bar{X}_j(2m)(j = 1, 2, \ldots, 200)$. Si $\tilde{\rho}_1(lk/2, 2m) < \tilde{\rho}_1(lk, m)$ ir al Paso 1c. En otro caso ir al Paso 2.

- Paso 1c. Dividir n_i en k bloques de tamaño $l \cdot m$. Calculamos la media de los bloques $\bar{\bar{X}}_j(k, lm)$ a partir de $\bar{X}_j(lm)(j = 1, 2, \ldots, k)$ y el intervalo de confianza estimado δ.
  Si $\delta/\bar{\bar{X}}_j(k, lm) < \gamma$ entonces construimos el intervalo de confianza para μ a partir de la ecuación (2.7) y paramos el algoritmo. En otro caso vamos al Paso 2.

```
RelationIndex = BatchesAccumulatorLength/FinalBatches;

for(i=0 to FinalBatches)
{
   for(j=i*RelationIndex to (i+1)*RelationIndex)
   {
      AuxM = AuxM + BinaryBatchesAccumulator[j];
      NItems++;
   }
   BatchesAccumulator[AccumulatorIndex] = AuxM/NItems;

   ...
}

for(j=0 to FinalBatches)
{
   x = BatchesAccumulator[j];
   AuxM = AuxM + x;
   AuxV = AuxV + x*x;
   NItems++;
}
```

```
FinalBatchMean = AuxM/NItems;
BatchVariance = AuxV/NItems-(FinalBatchMean*FinalBatchMean);
ConfidenceIntervalEstimate = StudentPercentil
                             *sqrt(BatchVariance/FinalBatches);

if(ConfidenceIntervalEstimate < FinalBatchMean*RelativePrecision)
{
  BatchEnded = true;
  go Paso 2
}

...
```

- Paso 2. Aumentar el número de iteraciones i+1 y aumentar el número de observaciones y volver al Paso 1a.

```
IterationCounter++;
CollectionSamples += StepSamples*pow(2.0,(IterationCounter/2));
BinaryBatchSamples = pow(2.0,((IterationCounter/2) + 1));

...
```

Como dijimos anteriormente, es necesario ajustar varios parámetros iniciales en el procedimiento de *batch means*:

- n_0: el número de observaciones del período transitorio inicial que han sido descartadas previamente (el valor por defecto es $n_0 = 800$). Este valor se obtiene aplicando previamente un procedimiento para la detección del período transitorio inicial. Hemos elegido el mismo valor que Law y Carson usan por defecto después de realizar varios experimentos (ver [3]).

- $1 - \alpha$: el nivel de confianza prefijado para los resultados finales $0 < \alpha < 1$. El valor por defecto es $\alpha = 0,1$.

- e_{max}: el valor máximo aceptable de la precisión relativa del intervalo de confianza ($0 < e_{max} < 1$; el valor por defecto es $e_{max} = 0,1$).

- c: umbral máximo de correlación; el valor por defecto es 0.4.

- γ: el criterio de parada mas ampliamente extendido para todos los métodos de análisis de la salida secuenciales, es la comprobación del error relativo de la mitad del intervalo de confianza a un determinado nivel de confianza. Para el método dicho error se calcula como, $\delta/\bar{\bar{X}}_j(k, lm)$. Si el error relativo calculado con la expresión anterior está por debajo de un umbral máximo $\gamma < \gamma_{max}$, entonces se da por finalizada la simulación. El valor por defecto de γ_{max} es 0,05 para un nivel de confianza aproximadamente del 95 %.

En general, elegir errores relativos pequeños es mejor que elegirlos grandes desde el punto de vista de la cobertura de las medias, ya que suele obligar a los métodos de análisis de la salida a simular más, con lo que disminuyen problemas de falta de normalidad y de correlación.

Los expertos Law y Carson explican que, debido al gran número de parámetros que se necesitan para especificar completamente el procedimiento, no es posible variar todos los parámetros al mismo tiempo. Debido a ello a partir de los resultados experimentales obtenidos, afirman que $l \cdot k = 400$ bloques con un tamaño $m = 2$, son suficientes para obtener un estimación de la correlación precisa. También indican que un valor del umbral de correlación $c = 0{,}4$ es razonable. Un valor más pequeño incrementará en gran medida el número de muestras requeridas.

3.3. Desarrollo de nuevos modelos de simulación en OMNeT++

Un modelo en OMNeT++ se construye a través de componentes o módulos que se comunican mediante intercambio de mensajes. Estos módulos pueden ser anidados, es decir, muchos módulos pueden ser agrupados para formar un módulo compuesto más complejo. Para poder desarrollar nuevos escenarios con el simulador OMNeT++ deben seguirse los siguientes pasos:

1. Definir la estructura del modelo en el lenguaje NED, ya sea con un editor de texto o con GNED, el editor gráfico de OMNeT++.
2. Los componentes activos del modelo (módulos simples) se programan en C++, usando las librerías de clases.
3. Definir un fichero ini. Este fichero de configuración puede describir muchas simulaciones con diferentes parámetros de entrada al sistema.

4. Construir el programa de simulación y ejecutarlo. Se enlazará el código con el núcleo de simulación de OMNeT++ y la interfaz de usuario que proporciona la herramienta, bien por línea de comandos o interfaz gráfica.

5. Los resultados de la simulación pueden escribirse en ficheros de salida. Pueden usarse las herramientas Plove y Scalars de OMNeT++ para visualizarlos.

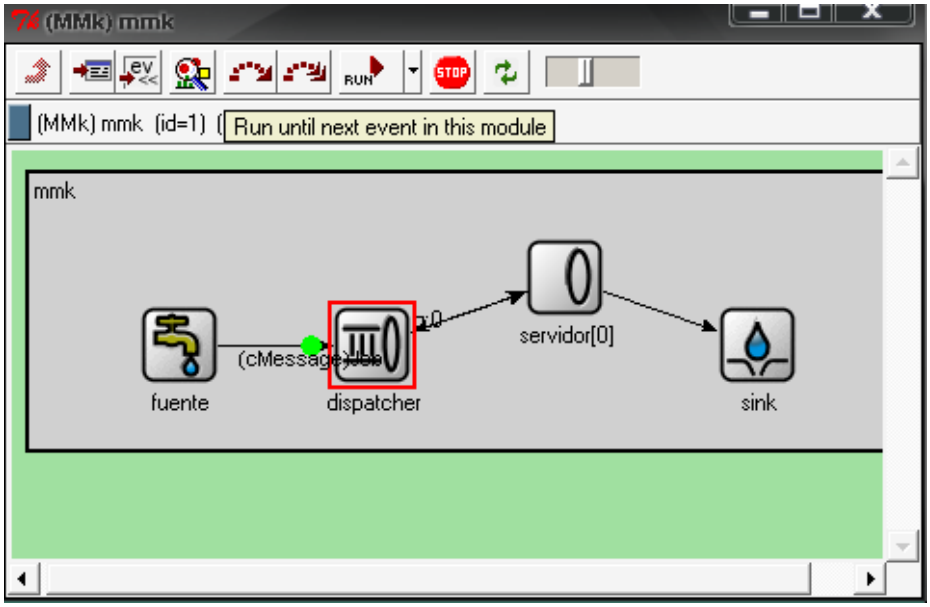

Figura 3.1: Vista en OMNeT++ de los diferentes módulos que forman el modelo

3.3.1. Modelos desarrollados

Para la demostración de la eficiencia del método *batch means* se han escogido dos modelos de simulación.

En primer lugar se ha implementado un sistema de colas M/M/k para realizar las simulaciones y observar el comportamiento del método al medir los tiempos de espera en cola y el tiempo de estancia en el sistema de los paquetes que genera una fuente de mensajes. El simulador OMNeT++ ya trae como ejemplo un sistema de colas M/M/1. El código puede encontrarse dentro de la carpeta omnetpp/samples/fifo o omnetpp/samples/queues. Este

código, sin embargo, está implementado mediante paso de mensajes asíncrono y utiliza el método activity() en lugar del recomendado handleMessage() que controla la simulación mediante sincronización de eventos[1]. En este proyecto se ha mejorado esta versión y además se ha añadido la posibilidad de simular el sistema hasta con k servidores.

Para el modelo de colas M/M/k se han diseñado un total de 4 clases en lenguaje C++: source.cc, server.cc, dispatcher.cc y sink.cc además de los respectivos ficheros .ned de cada módulo. Se han definido una fuente que genera paquetes cada cierto tiempo, una cola de almacenamiento, k servidores que ofrecen un determinado servicio y un sumidero al que llegan finalmente los mensajes y se eliminan.

En segundo lugar, se ha utilizado un modelo de ejemplo de OMNeT++ que consiste en una red de acceso compartido donde unas estaciones intentan comunicarse con un servidor mediante el protocolo Aloha. Este modelo también se puede encontrar en los ejemplos que trae OMNeT++. Puede localizarse en la carpeta omnetpp/samples/aloha. En este caso sólo tuvo que modificarse el código para usar la librería desarrollada que contiene nuestro procedimiento.

3.3.2. Ficheros NED

Cada entidad del modelo en la simulación necesita comunicarse a través de mensajes con ella misma y otras entidades. Estos mensajes se usan con varios propósitos. Unos representan paquetes o *jobs* que solicitan un servicio en las colas y otros almacenan información del estado de la entidad o se la envían a otras entidades.

En uno de los ejemplos que hemos utilizado para realizar las pruebas, encontramos 4 entidades diferentes: un módulo fuente, un módulo cola, un módulo servidor y un módulo sumidero o *sink*. El módulo fuente genera mensajes que representan paquetes de datos. Estos mensajes son enviados a la cola para ser enviados posteriormente en la medida de lo posible a un servidor libre. Finalmente el mensaje llega al *sink* donde se procesa y se elimina del sistema, recogiendo algunas estadísticas, como por ejemlo el tiempo de estancia en el sistema o el tiempo de espera en cola.

Todos estos módulos necesitan tener unos puertos de entrada y salida para enviar y recibir mensajes. El fichero .ned describe estos puertos. Por

[1]http://www.omnetpp.org/pmwiki/index.php?n=Main.HandleMessageVsActivity

ejemplo, el módulo servidor:

```
simple Server
    parameters:
       tservice : numeric,
       bitsPerSec : numeric;
    gates:
       in: fromDispatcher;
       out: toSink;
       out: toDispatcher;
endsimple
```

A partir de la versión 4.0, la sintaxis cambia ligeramente (ver más información en el documento [13] de cómo migrar desde versiones anteriores):

```
simple Server
{
    parameters:
        volatile double serviceTime @unit(s);
        volatile double bitsPerSec;
        @display("i=block/server");
    gates:
        input fromDispatcher;
        output toSink;
        output toDispatcher;
}
```

Además estos módulos pueden necesitar otros parámetros como por ejemplo, el tiempo de servicio. Estos parámetros también se definen en el fichero ned.

El último paso es unir todos los módulos en una red de trabajo o *network*. Esto se hace a través de lo que se denomina un módulo compuesto o *compound module*. Este módulo utiliza los módulos simples y los conecta entre ellos como si de una red se tratara, obteniendo como resultado un modelo más complejo y aproximado al modelo real que queremos estudiar. Por ejemplo en nuestro modelo, el aspecto de la sintaxis sería la siguiente:

```
module MMk
   ...

   submodules:
```

```
        source: Source;
        parameters: sendIaTime  = ta;
        display: "p=89,100;i=block/source";

        dispatcher : Dispatcher;
        parameters: numberOfServers = numK;
        gatesizes: toServer[numK], fromServer[numK];
        display: "p=189,100;i=block/queue;q=Queue";

        server: Server[numK];
        parameters: tservice = ts;
        display: "p=289,65,c;i=block/server";

        sink: Sink;
        gatesizes: in[numK];
        display: "p=389,100;i=block/sink";

    connections:

        source.out --> dispatcher.fromSource;
        for i=0..numK-1 do
            dispatcher.toServer[i] --> server[i].fromDispatcher,
            server[i].toDispatcher --> dispatcher.fromServer[i],
            server[i].toSink --> sink.in[i];
        endfor;

endmodule

network mmk : MMk
endnetwork
```

3.3.3. Ficheros C++

Una vez que hemos definido todas las entidades y hemos decidido como están conectadas entre sí, debemos decir a cada entidad cómo queremos que procese los mensajes. En términos concretos, la clase fuente genera mensajes cada cierto intervalo de tiempo o tasa de acuerdo a una cierta distribución de probabilidad, en este caso exponencial. Estos mensajes se envían a través del puerto de salida hacia la cola. Si echamos un vistazo al código del fichero source.cc,

```
void Source::handleMessage(cMessage *msg)
{
   ...

   cMessage *m = new cMessage("Job");
   m->setKind(TASK_FUENTE);
   send(m, "out");
   scheduleAt(simTime()+(double)par("sendIaTime"), sendMessageEvent);
}
```

Una vez el mensaje llega a la cola, éste es guardado en ella a la espera de que haya un servidor libre. Estos mensajes que llegan son tratados en el método handleMessage() de la clase Dispatcher. Cada vez que el manejador de eventos detecta que un mensaje llega a este módulo, el mensaje se guarda en la pila o *stack* y se ejecuta el método handleMessage(). Además, lo que se hace es comprobar de donde proviene el mensaje, bien de un servidor indicando que está libre (tipo `STATUS`), o bien de la fuente (tipo `TASK_FUENTE`). Cuando se detecta que hay un servidor libre se manda el primer mensaje almacenado (modelo FIFO) al servidor y se pone el estado del servidor como ocupado (`statusServer = 1`).

```
void Dispatcher::handleMessage(cMessage *msg)
{
   ...
   int index = 0;
   int type = msg->kind();
   int idServer = 0;

   switch(type)
   {

      case TASK_FUENTE:

         ...
         queue.insert( msg );
         users++;
         break;

      case STATUS:

         ...
```

```
            index = msg->par("id");
            statusServer[index] = msg->par("status");
            ...
            break;
    }

    if(!queue.empty())
    {

        idServer = getIdleServer();

        if (idServer!=-1)
        {

            cMessage *task  = (cMessage *) queue.getTail();
                ...
            statusServer[idServer]=1;
            task->addPar("id") = idServer;
            send(task,"toServer",idServer);
        }

    ...
    }
}
```

Cuando enviamos el mensaje al servidor, éste lo procesa durante un cierto tiempo de servicio y posteriormente lo envía al módulo Sink. La filosofía es la misma que en la clase Dispatcher. Al recibir el mensaje el módulo servidor el manejador de eventos ejecuta el método handleMessage() de la clase Server:

```
void Servidor::handleMessage(cMessage *msg)
{

   int type = msg->kind();

   switch (type) {

      case TASK_FUENTE:

         ...
         status = 1;
         idServer=msg->par("id");
         msgServiced = msg;
         tiempoServicio=par("tservice");
         scheduleAt(simTime() + timeService, taskIsDone);
         scheduleAt(simTime() + timeService, taskStatus);
         break;

      case TASK_DONE:

         ...
         send(msgServiced, "toSink");
         msgServiced=NULL;

      case STATUS:

         status = 0;
         statusReply = new cMessage("Status server");
         statusReply->setKind(STATUS);
         ...

         send(statusReply, "toDispatcher");

         ...
   }
}
```

Cuando se haya consumido el tiempo de servicio, se envía el mensaje al módulo Sink y se envía otro mensaje al módulo Dispatcher indicándole que estamos libres para recibir otro mensaje.

3.3.4. Ficheros ini

Estos ficheros son fundamentales para realizar la simulación. En ellos se describen los parámetros característicos de la misma, número de simulaciones a ejecutar, duración de la simulación, valor de los atributos de los módulos simples, ficheros donde se guardarán las estadísticas recogidas, etc.

```
[General]

network = mmk
num-rngs = 2
seed-0-mt=55
seed-1-mt=3

output-vector-file = mmk.vec
output-scalar-file = mmk.sca

[Cmdenv]
runs-to-execute = 1
express-mode = yes
...

[Run 1]
description="Sistema M/M/1 ts=0.1 ro=0.95"
mmk.numK = 1
mmk.ta = exponential(0.1052631,0)
mmk.ts = exponential(0.1,1)
```

En este caso se ha mostrado el fichero de configuración para un sistema M/M/1. Puede observarse como los parámetros característicos como son el tiempo de servicio y el tiempo entre llegadas se modelan siguiendo una distribución exponencial.

3.3.5. Generación de números aleatorios

Para cualquier simulación que utilice muestras aleatorias generadas por algún tipo de distribución estadística, debe tenerse en cuenta el siguiente aspecto. Los números aleatorios no se generan al azar. Por el contrario, se generan utilizando algoritmos deterministas. Los algoritmos toman el valor de una semilla generadora y realizan algunos cálculos deterministas sobre

ellas para producir un número aleatorio y la siguiente semilla. Tales algoritmos y sus implementaciones son llamados generadores de números aleatorios (en inglés RNGs) o, a veces, generadores de números pseudo-aleatorios o PRNGs para resaltar su carácter determinista. Los RNGs siempre producen la misma secuencia de números aleatorios. Esta es una propiedad útil y de gran importancia, porque hace que la simulación sea repetible.

Los RNGs producen números enteros uniformemente distribuidos in cierto rango, normalmente entre 0 o 1 y 2^{32}, y mediante tranformaciones matemáticas se generan variaciones aleatorias que se corresponden con distribuciones específicas como, por ejemplo, una distribución exponencial.

El simulador OMNeT++ está provisto de algunas clases para generar estos números aleatorios. Algunos ejemplos son Mersenne Twister o LCG (ver manual de usuario de OMNeT++ [15]), siendo la de Mersenne Twister la que mejores resultados presenta.

En los modelos utilizados para demostrar la eficiencia del mecanismo de finalización de la simulación, se han tenido en cuenta varias fuentes de números aleatorios para la correcta independencia entre las muestras recogidas en la ejecución de la simulación.

3.4. Uso de la clase cADByBatchMeans

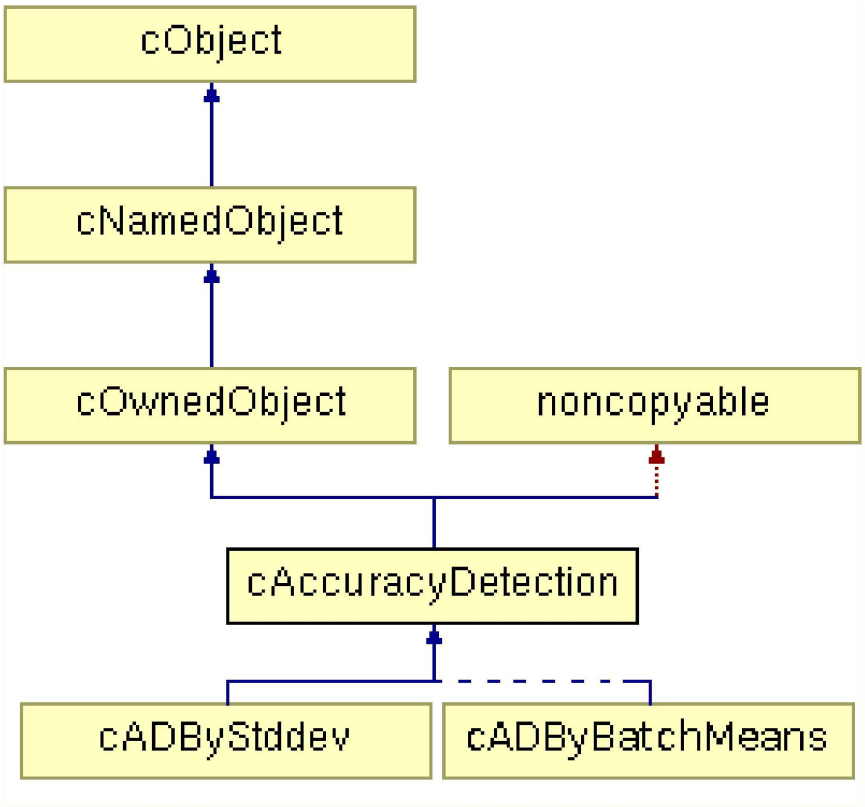

Figura 3.2: Jerarquía clase cADByBatchMeans

Esta librería no representa ninguna entidad, simplemente implementa la clase que se encargará de realizar la estimación de los datos para buscar un criterio de parada, mediante el método *batch means*. Para poner en funcionamiento el mecanismo de detección, siempre declararemos un objeto de esta clase cADByBatchMeans e invocaremos al método start(). Una vez el algoritmo finalice se mostrarán los resultados obtenidos, es decir, media obtenida, intervalo de confianza e iteraciones que se realizaron hasta finalizar la simulación. El valor por defecto del nivel de confianza del método es 90 %, aunque puede ser modificado por el usuario mediante la función cADByBatchMeans::setParameters(double ci, double cl).

Por ejemplo si quisiéramos analizar los tiempos de espera en la cola de un servidor, lo haríamos de la siguiente manera:

```
void Dispatcher::initialize()
{
   cStdDev WTimeStats;
   WTimeStats.setName("Queueing Time Stats (W)");

   ...

   cADByBatchMeans bm = new cADByBatchMeans(0.1,0.9);
   WTimeStats.addAccuracyDetection(bm);
   bm->start();

}
```

Hemos sobrecargado el método addAccuracyDetection ya definido en las librería de OMNeT++ para que así el usuario que está acostumbrado a trabajar con estos mecanismos no tenga que modificar en gran medida el código de sus clases.

Así por ejemplo, como lo que nos interesa son los tiempo de espera en cola, guardamos los valores que se van obteniendo en un objeto de la clase cStdDev, mediante el método cStdDev::collect(double val). Por tanto, por heredar la clase cADByBatchMeans a su vez de la clase cAccuracyDetection, cuando ejecutamos WTimeStats.collect(tespera), también se está ejecutando cADByBatchMeans::collect(tespera).

A medida que recogemos muestras, comprobaremos si el algoritmo ha encontrado una estimación válida que provoque la parada de la simulación por medio del método detected()

```
void Dispatcher::handleMessage(cMessage *msg)
{
   ...

   if (!queue.empty())
   {

      ...
      WTimeStats.collect(tespera);
```

```
      if (bm->detected())
      {
         bm->stop();
         endSimulation();
      }

   }
}
```

cuando el algoritmo haya recogido muestras suficientes que idiquen un resultado válido, se parará la ejecución y se llamará al método endSimulation() llamando automáticamente al método finish() del módulo en donde estemos usando el objeto de la clase cADByBatchMeans. Los resultados del algoritmo se muestran por la consola de la herramienta OMNeT++, al llamar al método PrintResults()

```
void Dispatcher::finish()
{

   bm->PrintResults();
   ...
}
```

Es importante comentar que la librería puede ser usadada tanto en la versión 3.x como en la 4.0 sin tener que hacer cambios en el código.

Capítulo 4

Pruebas

A continuación, mostramos los estudios realizados para probar la eficiencia del método *batch means*. Todas las pruebas se han realizado en la misma CPU con las siguientes características:

Versión OMNeT++	3.3
Procesador	Intel(R) Pentium(R) 4 CPU 1.80GHz
Memoria RAM	2 GB
Plataforma	Linux (Kernel version 2.6.18-3-686)

Cuadro 4.1: Entorno de pruebas

4.1. Sistema de colas M/M/1

Estos son los resultados obtenidos de simular un sistema de colas M/M/1 con distintos factores de utilización (ρ), desde el 5 % al 95 %, y observando los tiempos de espera en cola y de estancia en el sistema de los mensajes desde que se generan en la fuente, hasta que se destruyen. El tiempo de servicio de los mensajes sigue una distribución exponencial de media $t_s = 0{,}1s$. Se realizaron varias simulaciones, observando el comportamiento del método *batch means* al variar el nivel de confianza (n.c.) e intervalo de confianza (i.c.) exigidos.

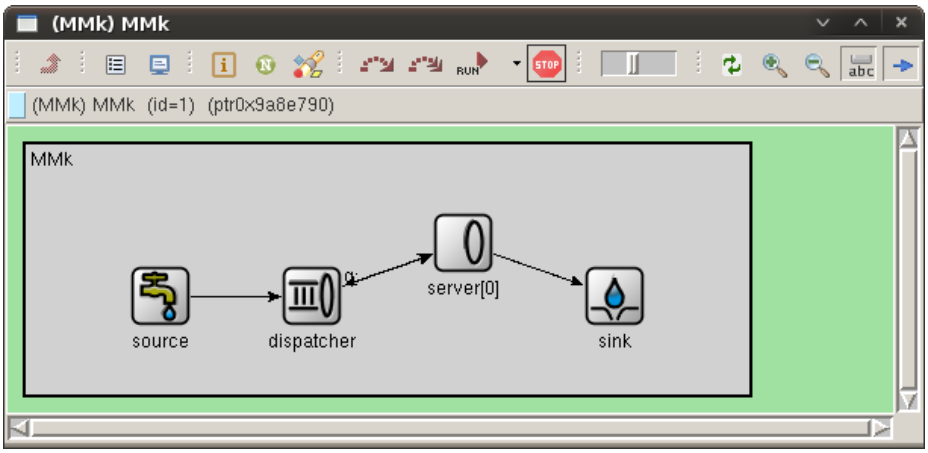

Figura 4.1: Vista en OMNeT++ de un modelo de colas M/M/1

Se puede demostrar analíticamente que para un sistema de colas M/M/1, el tiempo de espera en cola (W) es:

$$\bar{W} = \frac{\rho}{\mu \cdot (1 - \rho)} \tag{4.1}$$

siendo $\rho = \frac{\lambda}{\mu}$, es decir, la tasa de llegadas de mensajes entre la tasa de servicio de los mensajes, mientras que el tiempo de estancia en el sistema se calcula como:

$$\bar{T} = \bar{W} + t_s \tag{4.2}$$

Seguidamente se muestran los resultados de las diferentes simulaciones realizadas sobre el modelo de colas M/M/1:

ρ	**W teórico**	**W simulado**	**Intervalo Confianza**	**Rango**
0.05	0.0053	0.00534256	± 0.000252942	[0.00508961 , 0.0055955]
0.10	0.0111	0.0102805	± 0.000737874	[0.00954259 , 0.0110183]
0.15	0.0176	0.0166897	± 0.00126742	[0.0154223 , 0.0179571]
0.20	0.0250	0.0233773	± 0.00149073	[0.0218866 , 0.024868]
0.25	0.0333	0.0310229	± 0.00307306	[0.0279498 , 0.0340959]
0.30	0.0429	0.0414603	± 0.0041437	[0.0373166 , 0.045604]
0.35	0.0538	0.0511867	± 0.0050233	[0.0461634 , 0.05621]
0.40	0.0667	0.0631099	± 0.00602567	[0.0570843 , 0.0691356]
0.45	0.0818	0.0772943	± 0.00733082	[0.0699635 , 0.0846251]
0.50	0.1000	0.095822	± 0.00940596	[0.0864161 , 0.105228]
0.55	0.1222	0.110525	± 0.010338	[0.100187 , 0.120863]
0.60	0.1500	0.140972	± 0.0136084	[0.127363 , 0.15458]
0.65	0.1857	0.168109	± 0.0161587	[0.151951 , 0.184268]
0.70	0.2333	0.209792	± 0.0193119	[0.19048 , 0.229104]
0.75	0.3000	0.267352	± 0.0228166	[0.244536 , 0.290169]
0.80	0.4000	0.368301	± 0.0308184	[0.337483 , 0.399119]
0.85	0.5667	0.549767	± 0.0329339	[0.516833 , 0.582701]
0.90	0.9000	0.8657	± 0.0613267	[0.804373 , 0.927027]
0.95	1.9000	1.93342	± 0.121267	[1.81215 , 2.05469]

Cuadro 4.2: Sistema M/M/1: Tiempo espera en cola (W) con nivel de confianza del 90 %

Dichas simulaciones se realizaron ejecutando el simulador en modo consola y a velocidad `express`. A continuación se muestran los tiempos de ejecución de la simulación, tanto el real como el simulado, así como el número de iteraciones que realiza el procedimiento de *batch means* hasta llegar a una estimación válida del parámetro estudiado, en este caso el tiempo de espera en cola:

ρ	**Tiempo simulado**	**Tiempo ejecución**	**Iteraciones**
0.05	1h 13m	0.797s	12
0.10	~ 0	~ 0	9
0.15	~ 0	~ 0	7
0.20	~ 0	~ 0	7
0.25	~ 0	~ 0	6
0.30	~ 0	~ 0	5
0.35	~ 0	~ 0	5
0.40	~ 0	~ 0	5
0.45	~ 0	~ 0	5
0.50	~ 0	~ 0	5
0.55	~ 0	~ 0	6
0.60	~ 0	~ 0	6
0.65	~ 0	~ 0	7
0.70	~ 0	~ 0	8
0.75	~ 0	~ 0	9
0.80	32m 18s	0.381s	10
0.85	4h 20m	2.544s	15
0.90	6h 10m	4.808s	17
0.95	23h 1m	14.571s	20

Cuadro 4.3: Sistema M/M/1: Tiempo de ejecución e iteraciones con nivel de confianza del 90 %

Como vemos, a medida que la ocupación del sistema aumenta, el sistema tarda más tiempo en llegar a una cierta estabilidad en los valores de los tiempos de espera, de ahí que el algoritmo necesite recoger más muestras para obtener una estimación válida. Por ejemplo para el caso $\rho = 0{,}80$ el algoritmo necesita realizar 10 iteraciones, lo que implica que en la simulación se hayan recogido más de 50000 muestras para obtener un valor medio del tiempo de espera en cola.

La figura 4.2 muestra la comparación entre el valor teórico del tiempo de espera (W teórico) y el valor medio obtenido en la simulación (W simulado) con un nivel de confianza del 90 %.

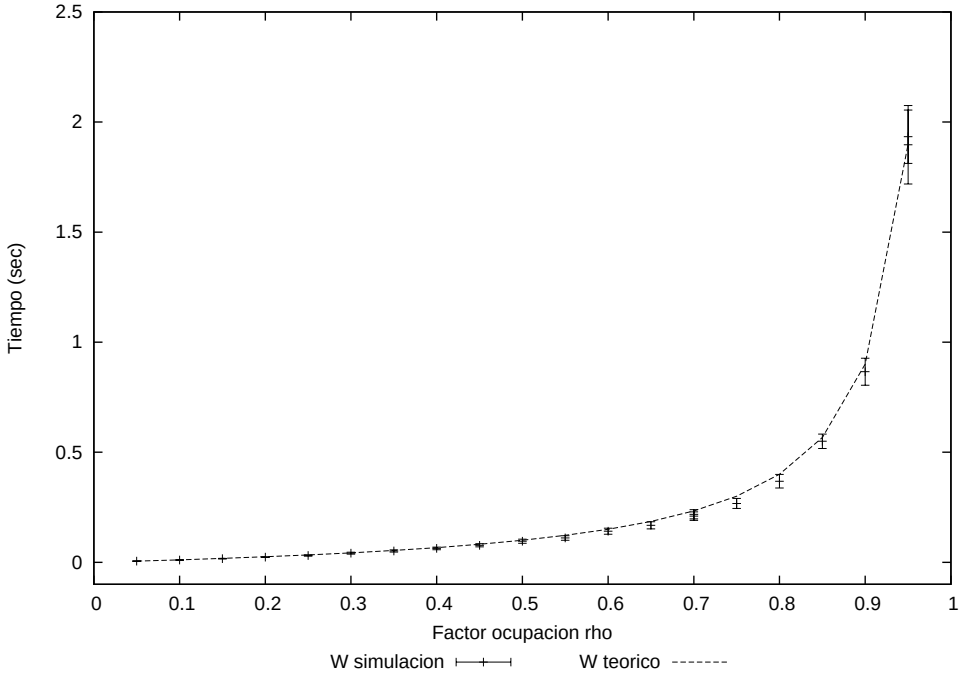

Figura 4.2: Sistema M/M/1: W teórico vs W simulado con nivel de confianza del 90 %

Como se observa, los resultados obtenidos en la simulación se ajustan bastante a los valores teóricos. Hay que destacar que para los valores $\rho = 0{,}70$ y $\rho = 0{,}95$, se calcularon los resultados utilizando dos semillas diferentes para generar las distribuciones de probabilidad, observándose cambios significativos en los valores del intervalo de confianza.

Seguidamente mostramos los resultados para un nivel de confianza más ajustado que el caso anterior. Se observa un aumento significativo lógico del número de iteraciones del algoritmo.

ρ	**W teórico**	**W simulado**	**Intervalo Confianza**	**Rango**
0.05	0.0053	0.0052811	± 0.000170586	[0.00511051 , 0.00545168]
0.10	0.0111	0.0110534	± 0.000502236	[0.0105512 , 0.0115557]
0.15	0.0176	0.0176208	± 0.000712218	[0.0169086 , 0.018333]
0.20	0.0250	0.0247357	± 0.000938809	[0.0237969 , 0.0256745]
0.25	0.0333	0.0327184	± 0.00133095	[0.0313874 , 0.0340493]
0.30	0.0429	0.0424472	± 0.00150663	[0.0409406 , 0.0439539]
0.35	0.0538	0.053182	± 0.00183745	[0.0513446 , 0.0550195]
0.40	0.0667	0.0655608	± 0.00220578	[0.063355 , 0.0677666]
0.45	0.0818	0.0799155	± 0.00278969	[0.0771258 , 0.0827052]
0.50	0.1000	0.0973826	± 0.00359525	[0.0937874 , 0.100978]
0.55	0.1222	0.118908	± 0.00473232	[0.114176 , 0.12364]
0.60	0.1500	0.141682	± 0.00573275	[0.135949 , 0.147414]
0.65	0.1857	0.178571	± 0.00788837	[0.170683 , 0.18646]
0.70	0.2333	0.222512	± 0.0108851	[0.211627 , 0.233398]
0.75	0.3000	0.296715	± 0.00901277	[0.287702 , 0.305728]
0.80	0.4000	0.389987	± 0.0134077	[0.37658 , 0.403395]
0.85	0.5667	0.54052	± 0.0214917	[0.519028 , 0.562011]
0.90	0.9000	0.914829	± 0.0455569	[0.869273 , 0.960386]
0.95	1.9000	1.88058	± 0.0726127	[1.80797 , 1.9532]

Cuadro 4.4: Sistema M/M/1: Tiempo espera en cola (W) con nivel de confianza del 95 %

Y estos son los tiempos de ejecución de la simulación y el número de iteraciones del algoritmo:

ρ	**Tiempo simulado**	**Tiempo ejecución**	**Iteraciones**
0.05	4h 10m	1.810s	14
0.10	3h 48m	1.113s	13
0.15	3h 27m	1.093s	13
0.20	3h 18m	1.110s	13
0.25	2h 24m	0.737s	12
0.30	3h 5m	1.093s	13
0.35	3h 24m	1.119s	13
0.40	2h 58m	1.108s	13
0.45	2h 38m	1.095s	13
0.50	2h 22m	1.182s	13
0.55	2h 9m	1.049s	13
0.60	1h 19m	0.738s	12
0.65	1h 49m	1.089s	13
0.70	2h 50m	1.806s	14
0.75	7h 25m	4.944s	17
0.80	6h 58m	4.851s	17
0.85	6h 33m	4.844s	17
0.90	12h 20m	9.591s	19
0.95	2d 23h	58.954s	24

Cuadro 4.5: Sistema M/M/1: Tiempo de ejecución e iteraciones con nivel de confianza del 95 %

Hay que destacar que el algoritmo llega a realizar hasta 24 iteraciones. Sin embargo, a la velocidad máxima que puede ejecutarse la herramienta, supone un tiempo mínimo de espera para el observador hasta obtener los resultados.

Nuevamente la figura 4.3 muestra la comparación entre el valor teórico del tiempo de espera y el valor medio obtenido en la simulación con el nivel de confianza del 95 %.

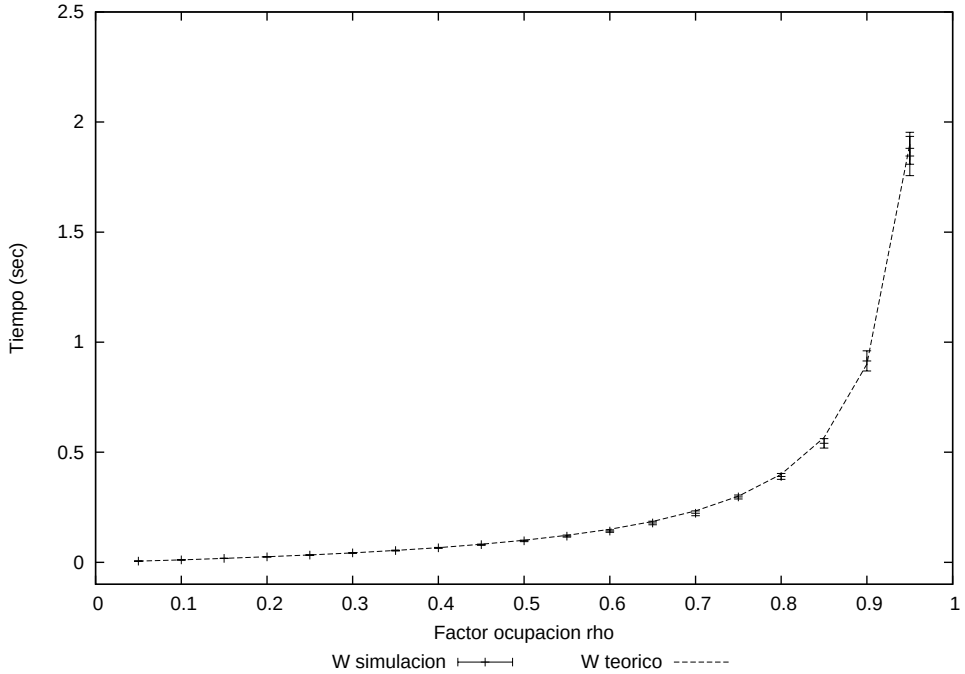

Figura 4.3: Sistema M/M/1: W teórico vs W simulado con nivel de confianza del 95 %

En este caso, los valores simulados están aún más ajustados a los valores teóricos comparado con el caso anterior donde el nivel de confianza era el 90 %.

La figura 4.4 muestra la diferencia del número de iteraciones que realiza el algoritmo al variar el nivel de confianza entre el 90 % y 95 %.

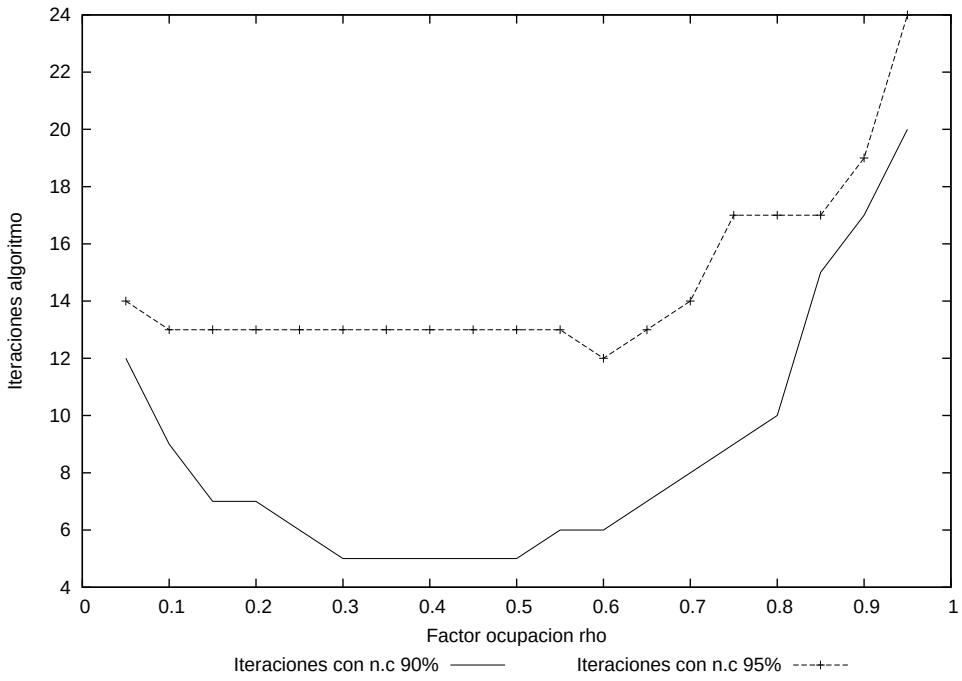

Figura 4.4: Sistema M/M/1: Comparativa iteraciones para el cálculo de W

Como era de esperar, para el caso de nivel de confianza del 95 % el número de iteraciones hasta llegar a un resultado válido es mayor, y crece a medida que la ocupación en el sistema es mayor, puesto que hay más valores que analizar.

Para completar el estudio, en este primer modelo, se observó además el tiempo de estancia en el sistema de los mensajes generados en el sistema M/M/1, produciéndose los siguientes resultados:

ρ	**T teórico**	**T simulado**	**Intervalo Confianza**	**Rango**
0.05	0.1053	0.102787	± 0.00383403	[0.0989526 , 0.106621]
0.10	0.1111	0.108105	± 0.00436786	[0.103738 , 0.112473]
0.15	0.1176	0.127432	± 0.0105858	[0.116846 , 0.138017]
0.20	0.1250	0.136811	± 0.0118453	[0.124966 , 0.148656]
0.25	0.1333	0.135804	± 0.00977512	[0.126029 , 0.145579]
0.30	0.1429	0.137806	± 0.00802673	[0.129779 , 0.145833]
0.35	0.1538	0.14771	± 0.00964246	[0.138067 , 0.157352]
0.40	0.1667	0.15571	± 0.00950111	[0.146208 , 0.165211]
0.45	0.1818	0.167529	± 0.0111859	[0.156343 , 0.178715]
0.50	0.2000	0.184025	± 0.0107333	[0.173292 , 0.194758]
0.55	0.2222	0.200915	± 0.0126952	[0.18822 , 0.21361]
0.60	0.2500	0.234162	± 0.00818686	[0.225975 , 0.242349]
0.65	0.2857	0.265599	± 0.0108309	[0.254768 , 0.27643]
0.70	0.3333	0.299356	± 0.0147618	[0.284594 , 0.314118]
0.75	0.4000	0.396075	± 0.0237569	[0.372318 , 0.419832]
0.80	0.5000	0.472323	± 0.0282108	[0.444112 , 0.500533]
0.85	0.6667	0.624047	± 0.0351947	[0.588852 , 0.659241]
0.90	1.0000	0.926424	± 0.0604711	[0.865953 , 0.986895]
0.95	2.0000	1.99694	± 0.17849	[1.81845 , 2.17543]

Cuadro 4.6: Sistema M/M/1: Tiempo estancia en el sistema (T) con nivel de confianza del 90 %

Siendo los tiempos de ejecución de la simulación y el número de iteraciones del algoritmo los siguientes:

ρ	**Tiempo simulado**	**Tiempo ejecución**	**Iteraciones**
0.05	~ 0	~ 0	3
0.10	~ 0	~ 0	3
0.15	~ 0	~ 0	1
0.20	~ 0	~ 0	1
0.25	~ 0	~ 0	2
0.30	~ 0	~ 0	3
0.35	~ 0	~ 0	3
0.40	~ 0	~ 0	4
0.45	~ 0	~ 0	4
0.50	~ 0	~ 0	5
0.55	~ 0	~ 0	5
0.60	~ 0	~ 0	8
0.65	~ 0	~ 0	8
0.70	~ 0	~ 0	9
0.75	31m 35s	0.341s	11
0.80	59m 42s	0.727s	12
0.85	2h 20m	1.764s	14
0.90	4h 25m	3.503s	16
0.95	17h 58m	14.421s	20

Cuadro 4.7: Sistema M/M/1: Tiempo de ejecución e iteraciones con nivel de confianza del 90 %

Para una carga del sistema baja, el método encuentra una estimación muy rápidamente. El sistema se encuentra poco congestionado, y llega antes a un estado estacionario.

La figura 4.5 muestra la comparación entre el valor teórico del tiempo T y el valor medio obtenido en la simulación con el intervalo de confianza deseado.

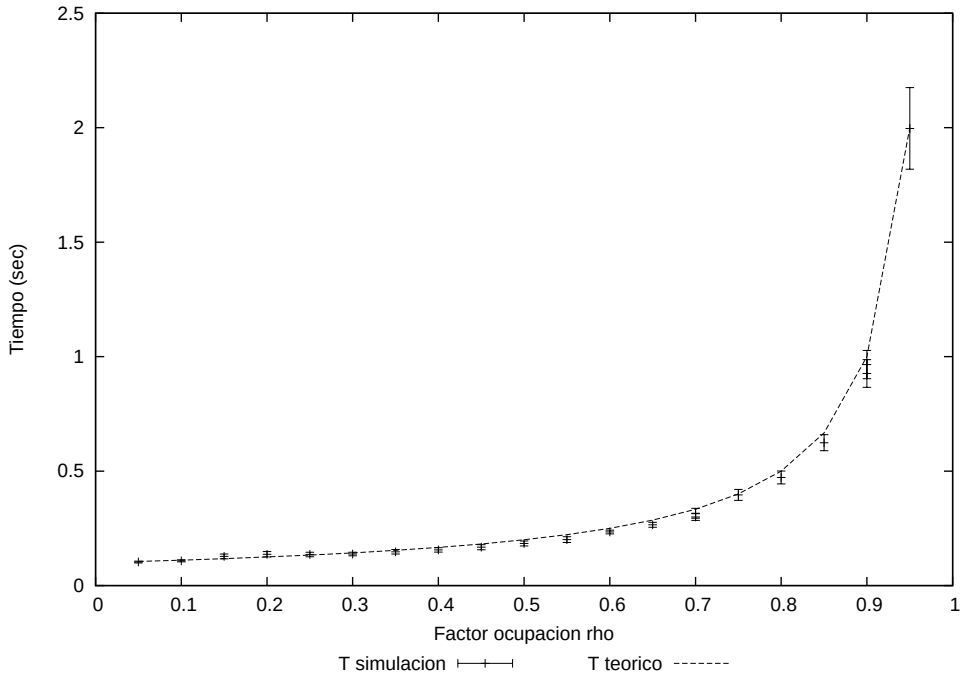

Figura 4.5: Sistema M/M/1: T teórico vs T simulado con nivel de confianza del 90 %

En esta ocasión también se han usado dos semillas diferentes para $\rho = 0{,}70$ y $\rho = 0{,}90$, y los resultados obtenidos se ajustan más al valor teórico.

Y a continuación, se muestran los resultados para el tiempo T pero con un nivel de confianza mayor que en el caso anterior:

ρ	**T teórico**	**T simulado**	**Intervalo Confianza**	**Rango**
0.05	0.1053	0.103781	± 0.00496751	[0.0988135 , 0.108749]
0.10	0.1111	0.105918	± 0.00453758	[0.101381 , 0.110456]
0.15	0.1176	0.111347	± 0.00518814	[0.106159 , 0.116535]
0.20	0.1250	0.12184	± 0.00535608	[0.116484 , 0.127196]
0.25	0.1333	0.130276	± 0.00605578	[0.12422 , 0.136332]
0.30	0.1429	0.139063	± 0.00694437	[0.132119 , 0.146008]
0.35	0.1538	0.148699	± 0.00686889	[0.14183 , 0.155568]
0.40	0.1667	0.159904	± 0.00797091	[0.151933 , 0.167874]
0.45	0.1818	0.173906	± 0.00631029	[0.167595 , 0.180216]
0.50	0.2000	0.189391	± 0.00774346	[0.181647 , 0.197134]
0.55	0.2222	0.208246	± 0.00968095	[0.198565 , 0.217927]
0.60	0.2500	0.253498	± 0.0119175	[0.24158 , 0.265415]
0.65	0.2857	0.285063	± 0.00790979	[0.277153 , 0.292973]
0.70	0.3333	0.33098	± 0.00819134	[0.322789 , 0.339171]
0.75	0.4000	0.384909	± 0.0189431	[0.365966 , 0.403852]
0.80	0.5000	0.490617	± 0.016599	[0.474018 , 0.507216]
0.85	0.6667	0.664335	± 0.0259065	[0.638428 , 0.690241]
0.90	1.0000	1.01511	± 0.0457561	[0.969359 , 1.06087]
0.95	2.0000	1.98054	± 0.0726586	[1.90788 , 2.0532]

Cuadro 4.8: Sistema M/M/1: Tiempo estancia en el sistema (T) con nivel de confianza del 95 %

Siendo los tiempos de ejecución de la simulación y el número de iteraciones del algoritmo los siguientes:

ρ	**Tiempo simulado**	**Tiempo ejecución**	**Iteraciones**
0.05	~ 0	~ 0	3
0.10	~ 0	~ 0	4
0.15	~ 0	~ 0	4
0.20	~ 0	~ 0	5
0.25	~ 0	~ 0	5
0.30	~ 0	~ 0	5
0.35	~ 0	~ 0	6
0.40	~ 0	~ 0	6
0.45	~ 0	~ 0	7
0.50	~ 0	~ 0	7
0.55	~ 0	~ 0	7
0.60	39m 30s	0.389s	11
0.65	3h 3m	1.821s	14
0.70	7h 56m	5.276s	17
0.75	1h 35m	1.078s	13
0.80	6h 57m	4.946s	17
0.85	9h 48m	7.302s	18
0.90	12h 20m	9.639s	19
0.95	2d 23h	58.823s	24

Cuadro 4.9: Sistema M/M/1: Tiempo de ejecución e iteraciones con nivel de confianza del 95 %

A continuación, la figura 4.6 muestra la comparación entre el valor teórico del tiempo en el sistema T y el valor medio obtenido en la simulación con el intervalo de confianza deseado.

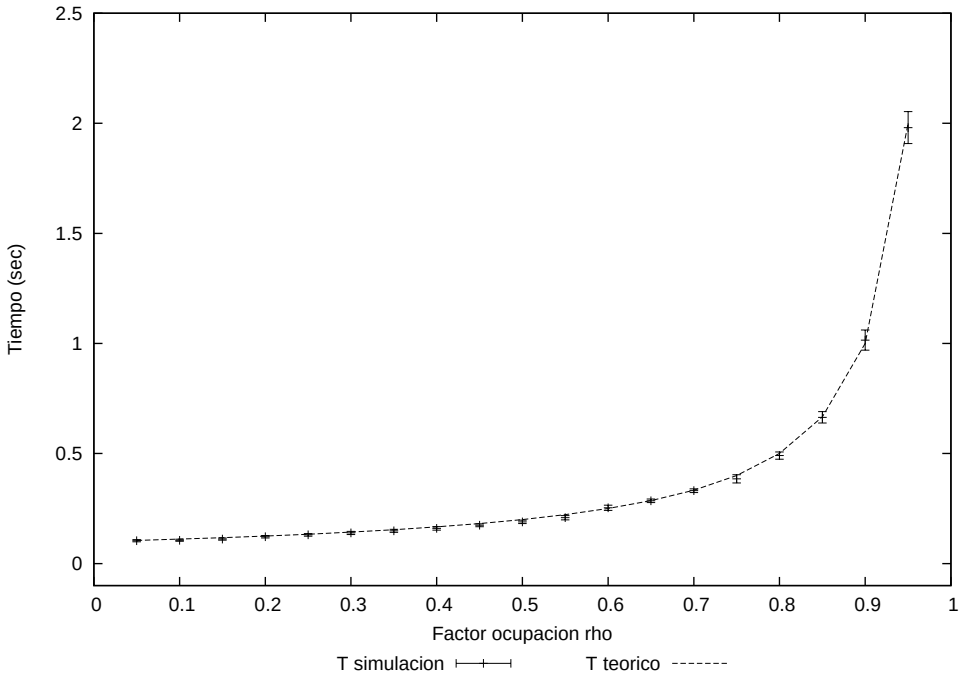

Figura 4.6: Sistema M/M/1: T teórico vs T simulado con nivel de confianza del 95 %

Por último, la figura 4.7 muestra la diferencia del número de iteraciones que realiza el algoritmo al variar el nivel de confianza exigido al método *batch means*.

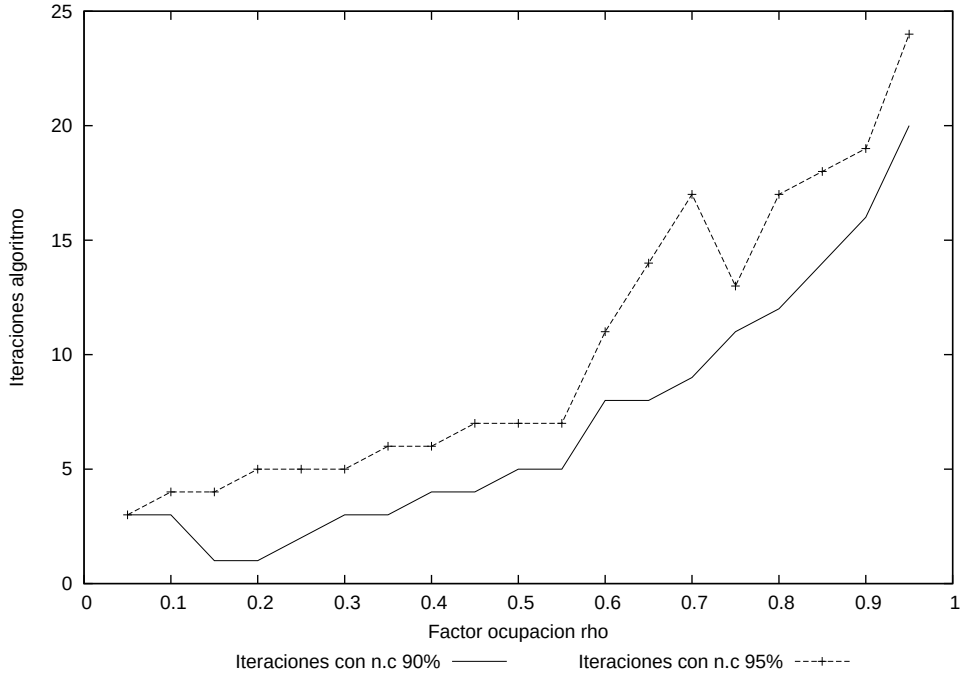

Figura 4.7: Sistema M/M/1: Comparativa iteraciones para el cálculo de T

Con este último ejemplo se ha querido demostrar que se pueden obtener un mecanismo de control del final de la simulación independientemente del parámetro o variable de salida a estudiar, que usemos para determinar el final de la simulación con el método de *batch means*.

4.2. Sistema de colas M/M/2

En esta ocasión, mostramos los resultados obtenidos de simular un sistema de colas M/M/2 con distintos factores de utilización, desde el 5 % al 95 %, y observando los tiempos de espera en cola y de estancia en el sistema de los mensajes que se reciben. El tiempo de servicio de los mensajes sigue una distribución exponencial de media $t_s = 0{,}1s$, pero esta vez con dos servidores para recibir los posibles mensajes generados por la fuente. Se realizaron varias simulaciones, observando el comportamiento del método *batch means* al variar el nivel de confianza (n.c.) exigido.

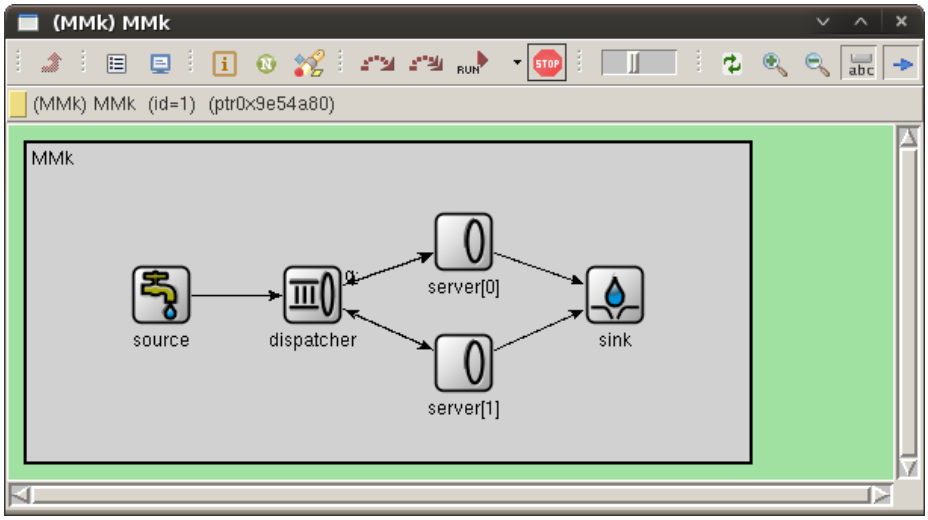

Figura 4.8: Vista en OMNeT++ de un modelo de colas M/M/2

Como en el caso anterior, se puede demostrar analíticamente que para un sistema de colas M/M/2, el tiempo de espera en cola (W) es:

$$\bar{W} = \left[\frac{I^2\rho}{2(1-\rho)^2}\right] \cdot \left[1 + I + \frac{I^2}{2(1-\rho)}\right]^{-1} \tag{4.3}$$

siendo $\rho = \frac{\lambda}{2\mu}$, mientras que el tiempo de estancia en el sistema se calcula como:

$$\bar{T} = \bar{W} + t_s \tag{4.4}$$

Seguidamente se muestran los resultados de las diferentes simulaciones realizadas sobre el modelo de colas M/M/2:

ρ	**W teórico**	**W simulado**	**Intervalo Confianza**	**Rango**
0.05	0.00025	0.000269482	± 2.68744e-05	[0.000242608 , 0.000296357]
0.10	0.00101	0.000994995	± 6.78149e-05	[0.00092718 , 0.00106281]
0.15	0.00230	0.0023028	± 0.000205627	[0.00209717 , 0.00250842]
0.20	0.00417	0.00445937	± 0.000436164	[0.0040232 , 0.00489553]
0.25	0.00667	0.00734718	± 0.00071912	[0.00662806 , 0.0080663]
0.30	0.00989	0.00841215	± 0.000716156	[0.00769599 , 0.0091283]
0.35	0.01396	0.0137332	± 0.00136413	[0.0123691 , 0.0150973]
0.40	0.01905	0.0204158	± 0.00150439	[0.0189114 , 0.0219201]
0.45	0.02539	0.0288107	± 0.00280122	[0.0260095 , 0.0316119]
0.50	0.03333	0.034877	± 0.00306347	[0.0318135 , 0.0379404]
0.55	0.04337	0.0403565	± 0.00352839	[0.0368281 , 0.0438849]
0.60	0.05625	0.0571668	± 0.00456645	[0.0526003 , 0.0617332]
0.65	0.07316	0.0816856	± 0.00803805	[0.0736476 , 0.0897237]
0.70	0.09608	0.0934175	± 0.00758596	[0.0858316 , 0.101003]
0.75	0.12857	0.132386	± 0.0123222	[0.120064 , 0.144708]
0.80	0.17778	0.165383	± 0.0111896	[0.154193 , 0.176572]
0.85	0.26036	0.262828	± 0.0218342	[0.240994 , 0.284662]
0.90	0.42632	0.321162	± 0.0271144	[0.294048 , 0.348276]
0.95	0.92564	0.86858	± 0.070188	[0.798392 , 0.938768]

Cuadro 4.10: Sistema M/M/2: Tiempo espera en cola (W) con nivel de confianza del 90 %

Siendo los tiempos de ejecución de la simulación y el número de iteraciones del algoritmo los siguientes:

ρ	**Tiempo simulado**	**Tiempo ejecución**	**Iteraciones**
0.05	1d 3h	2.582s	15
0.10	9h 53m	1.924s	14
0.15	1h 19s	0.388s	10
0.20	~ 0	~ 0	8
0.25	~ 0	~ 0	9
0.30	~ 0	~ 0	8
0.35	~ 0	~ 0	7
0.40	~ 0	~ 0	9
0.45	~ 0	~ 0	9
0.50	~ 0	~ 0	9
0.55	~ 0	~ 0	9
0.60	~ 0	~ 0	8
0.65	~ 0	~ 0	9
0.70	17m 5s	0.396s	10
0.75	16m 8s	0.405s	11
0.80	15m 5s	0.4s	10
0.85	1h 10m	1.842s	14
0.90	39m 55s	1.119s	13
0.95	9h	14.723s	20

Cuadro 4.11: Sistema M/M/2: Tiempo de ejecución e iteraciones con nivel de confianza del 90 %

Seguidamente la figura 4.9 muestra la comparación entre el valor teórico del tiempo de espera en cola W y el valor medio obtenido en la simulación con el intervalo de confianza deseado:

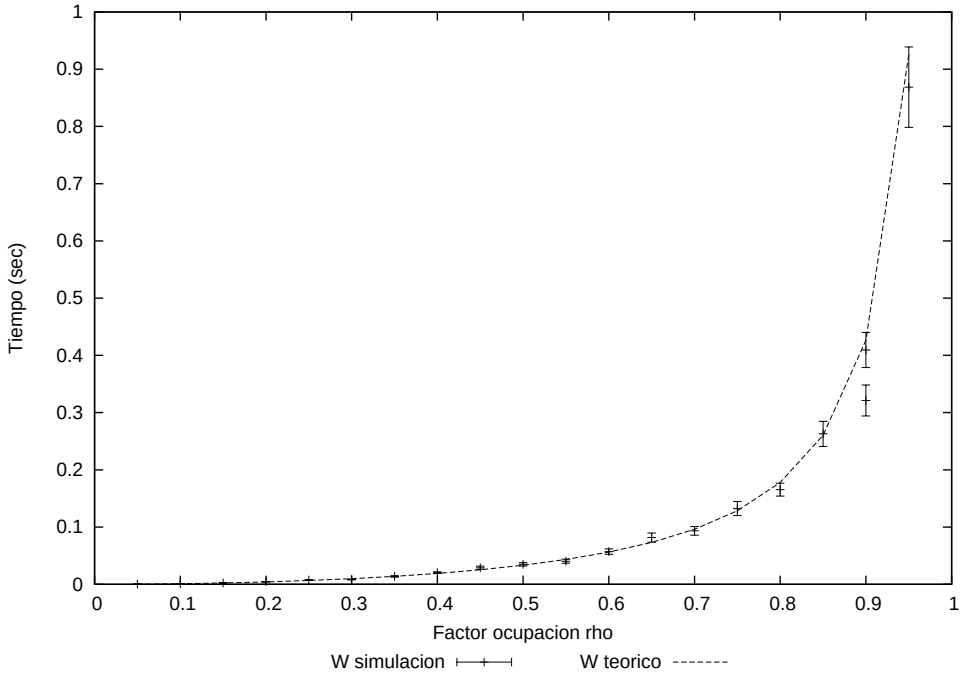

Figura 4.9: Sistema M/M/2: W teórico vs W simulado con nivel de confianza del 90 %

En este caso, se puede apreciar mucho más claramente el efecto de utilizar dos semillas diferentes para generar las distribuciones de probabilidad en el modelo estudiado. Si nos fijamos en el valor de $rho = 0{,}9$, se observa un cambio sustancial en el valor medio del tiemo de espera en cola.

Nuevamente los resultados del tiempo de espera en cola (W) pero con un nivel de confianza mayor:

ρ	**W teórico**	**W simulado**	**Intervalo Confianza**	**Rango**
0.05	0.00025	0.000255061	± 1.24323e-05	[0.000242628 , 0.000267493]
0.10	0.00101	0.00104022	± 4.66966e-05	[0.000993525 , 0.00108692]
0.15	0.00230	0.00229253	± 4.5958e-05	[0.00224657 , 0.00233849]
0.20	0.00417	0.00421408	± 0.000156111	[0.00405796 , 0.00437019]
0.25	0.00667	0.00675158	± 0.00033667	[0.00641491 , 0.00708825]
0.30	0.00989	0.00963027	± 0.000449757	[0.00918051 , 0.01008]
0.35	0.01396	0.013623	± 0.000515364	[0.0131076 , 0.0141383]
0.40	0.01905	0.0195849	± 0.000924364	[0.0186606 , 0.0205093]
0.45	0.02539	0.0252825	± 0.000850675	[0.0244318 , 0.0261332]
0.50	0.03333	0.0330874	± 0.00159986	[0.0314876 , 0.0346873]
0.55	0.04337	0.0421302	± 0.00132302	[0.0408071 , 0.0434532]
0.60	0.05625	0.0560491	± 0.00268598	[0.0533631 , 0.0587351]
0.65	0.07316	0.0708512	± 0.00335275	[0.0674985 , 0.074204]
0.70	0.09608	0.0966462	± 0.00347131	[0.0931749 , 0.100118]
0.75	0.12857	0.123427	± 0.0059939	[0.117434 , 0.129421]
0.80	0.17778	0.171966	± 0.00788229	[0.164083 , 0.179848]
0.85	0.26036	0.2615	± 0.00856941	[0.252931 , 0.27007]
0.90	0.42632	0.410039	± 0.0198471	[0.390192 , 0.429886]
0.95	0.92564	0.879032	± 0.0416207	[0.837411 , 0.920652]

Cuadro 4.12: Sistema M/M/2: Tiempo espera en cola (W) con nivel de confianza del 95 %

Observamos que en este caso el número de iteraciones es grande, pero la simulación no acarrea un coste temporal muy significativo.

Los tiempos de ejecución de la simulación y el número de iteraciones del algoritmo son los siguientes:

ρ	**Tiempo simulado**	**Tiempo ejecución**	**Iteraciones**
0.05	7d 2h	15.017s	20
0.10	19h 51m	3.423s	16
0.15	1d 12h	9.976s	19
0.20	9h 56m	3.526s	16
0.25	2h 21m	1.191s	13
0.30	3h 19m	1.782s	14
0.35	3h 58m	2.562s	15
0.40	1h 30m	1.217s	13
0.45	2h 12m	1.899s	14
0.50	1h 10m	1.145s	13
0.55	3h 36m	3.620s	16
0.60	1h 39m	1.840s	14
0.65	2h 8m	2.530s	15
0.70	1h 58m	2.606s	15
0.75	1h 19m	1.862s	14
0.80	2h 29s	3.561s	16
0.85	10h 2m	15.058s	20
0.90	6h 11m	9.889s	19
0.95	18h	29.607s	22

Cuadro 4.13: Sistema M/M/2: Tiempo de ejecución e iteraciones con nivel de confianza del 95 %

En esta caso, el número de iteraciones del método aumenta. Nuevamente el ajuste del intervalo de confianza, provoca que los resultados obtenidos sean más precisos a costa de perder más tiempo en alcanzar el final de la simulación.

A continuación, la figura 4.10 muestra la comparación entre el valor teórico del tiempo de espera en cola W y el valor medio obtenido en la simulación con el intervalo de confianza deseado.

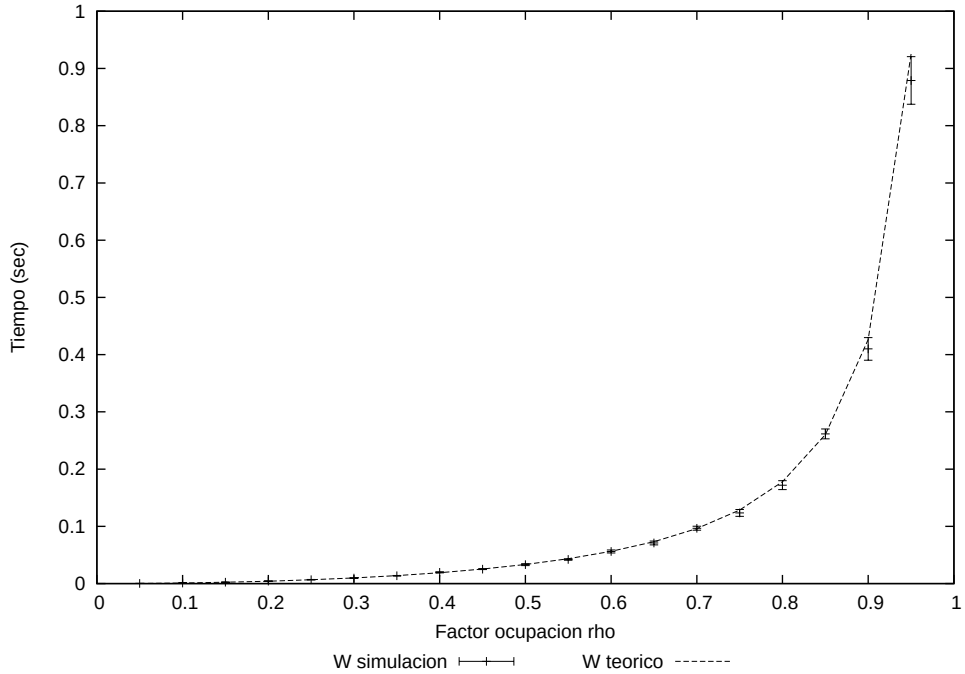

Figura 4.10: Sistema M/M/2: W teórico vs W simulado con nivel de confianza del 95 %

Como se observa, los resultados son más ajustados, el intervalo de confianza en cada uno de los puntos es mucho menor. Hay que recordar de nuevo que dependiendo de la semilla utilizada podríamos obtener resultados diferentes.

En la figura 4.11 se muestra la diferencia del número de iteraciones que realiza el algoritmo al variar el nivel de confianza exigido al método *batch means*.

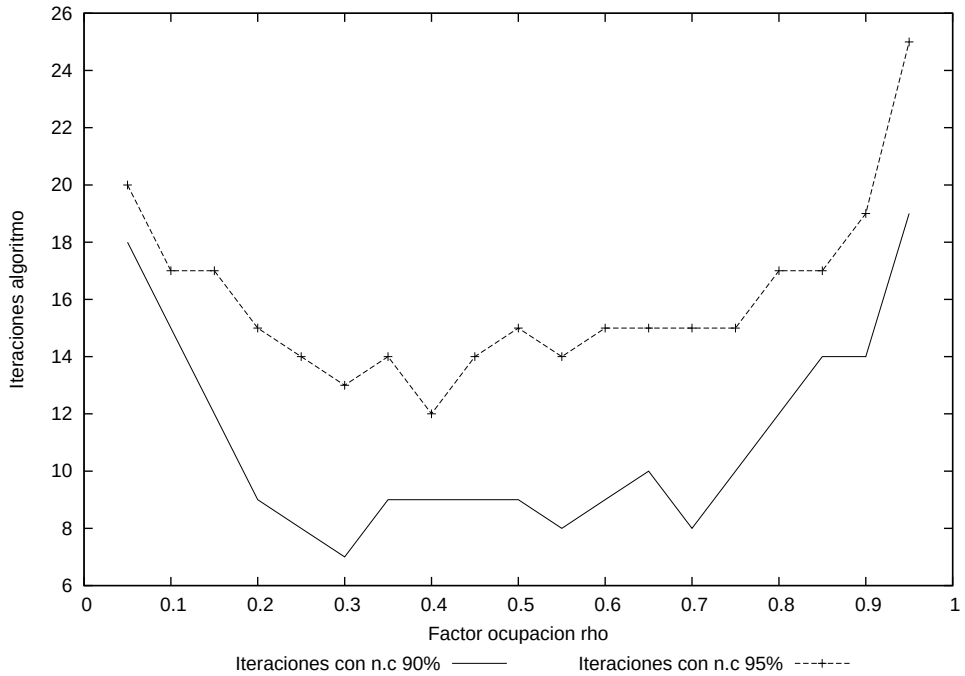

Figura 4.11: Sistema M/M/2: Comparativa iteraciones para el cálculo de W

Como en los casos anteriores, el número de iteraciones para el intervalo de confianza del 95 % es mayor que en el caso del 90 % y aumenta a medida que el sistema está más cargado.

Por último, los resultados del tiempo de estancia en el sistema (T) al igual que en el modelo anterior:

ρ	**T teórico**	**T simulado**	**Intervalo Confianza**	**Rango**
0.05	0.10025	0.0963016	± 0.00479736	[0.0915042 , 0.101099]
0.10	0.10101	0.106316	± 0.00482035	[0.101495 , 0.111136]
0.15	0.10230	0.103126	± 0.00523578	[0.0978904 , 0.108362]
0.20	0.10417	0.0964264	± 0.004834	[0.0915924 , 0.10126]
0.25	0.10667	0.102409	± 0.0035697	[0.0988398 , 0.105979]
0.30	0.10989	0.107468	± 0.00396134	[0.103507 , 0.111429]
0.35	0.11396	0.11406	± 0.00771415	[0.106346 , 0.121774]
0.40	0.11905	0.112114	± 0.00431787	[0.107796 , 0.116431]
0.45	0.12539	0.12674	± 0.0056016	[0.121139 , 0.132342]
0.50	0.13333	0.128458	± 0.00746214	[0.120996 , 0.13592]
0.55	0.14337	0.127584	± 0.00622552	[0.121358 , 0.13381]
0.60	0.15625	0.162186	± 0.00999416	[0.152192 , 0.17218]
0.65	0.17316	0.183312	± 0.00881011	[0.174502 , 0.192122]
0.70	0.19608	0.192825	± 0.00815057	[0.184675 , 0.200976]
0.75	0.22857	0.233313	± 0.0124447	[0.220868 , 0.245757]
0.80	0.27778	0.269735	± 0.0118912	[0.253772 , 0.277554]
0.85	0.36036	0.363107	± 0.0221584	[0.340949 , 0.385266]
0.90	0.52632	0.419852	± 0.0274672	[0.392385 , 0.447319]
0.95	1.02564	0.968589	± 0.0703176	[0.898271 , 1.03891]

Cuadro 4.14: Sistema M/M/2: Tiempo estancia en el sistema (T) con nivel de confianza del 90 %

Siendo los tiempos de ejecución de la simulación y el número de iteraciones del algoritmo los siguientes:

ρ	**Tiempo simulado**	**Tiempo ejecución**	**Iteraciones**
0.05	~ 0	~ 0	1
0.10	~ 0	~ 0	1
0.15	~ 0	~ 0	1
0.20	~ 0	~ 0	1
0.25	~ 0	~ 0	2
0.30	~ 0	~ 0	3
0.35	~ 0	~ 0	2
0.40	~ 0	~ 0	1
0.45	~ 0	~ 0	4
0.50	~ 0	~ 0	4
0.55	~ 0	~ 0	5
0.60	~ 0	~ 0	6
0.65	~ 0	~ 0	9
0.70	17m 5s	0.386s	10
0.75	16m 8s	0.380s	11
0.80	15m 5s	0.375s	10
0.85	1h 10m	1.794s	14
0.90	39m 55s	1.052s	13
0.95	9h	14.644s	20

Cuadro 4.15: Sistema M/M/2: Tiempo de ejecución e iteraciones con nivel de confianza del 90 %

Nuevamente la figura 4.12 muestra la comparación entre el valor teórico del tiempo en el sistema T y el valor medio obtenido en la simulación con el intervalo de confianza deseado.

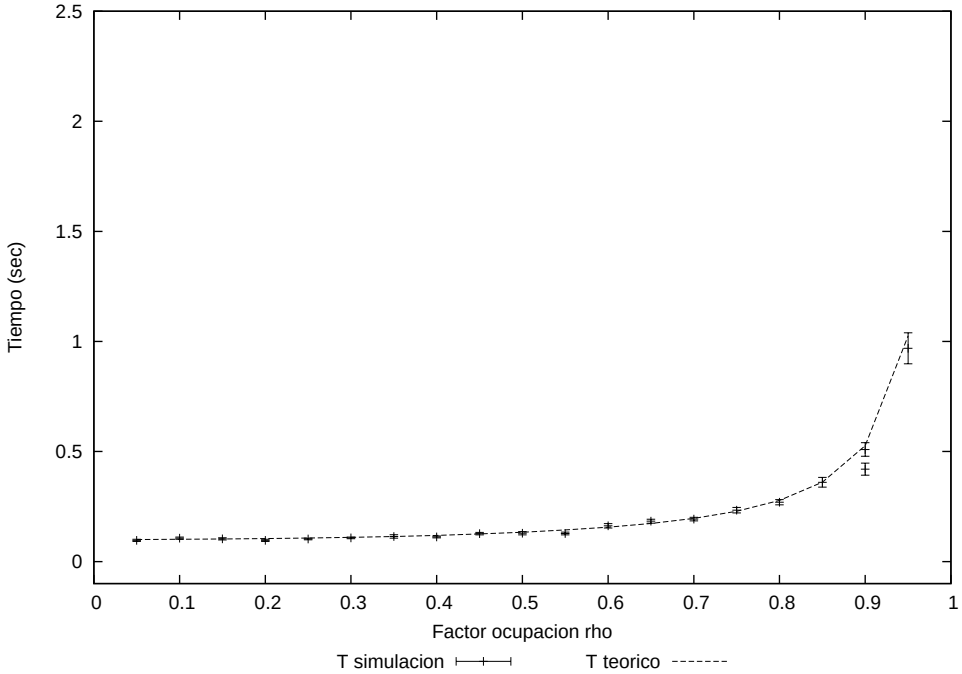

Figura 4.12: Sistema M/M/2: T teórico vs T simulado con nivel de confianza del 90 %

También en este caso, se aprecia la diferencia del valor de la semilla utilizada en el caso $rho = 0{,}9$.

Y finalmente, los resultados del tiempo en el sistema T para un nivel de confianza mayor:

ρ	**T teórico**	**T simulado**	**Intervalo Confianza**	**Rango**
0.05	0.10025	0.0951654	± 0.00411585	[0.0910495 , 0.0992812]
0.10	0.10101	0.0974886	± 0.00442285	[0.0930658 , 0.101912]
0.15	0.10230	0.100042	± 0.00458698	[0.0954554 , 0.104629]
0.20	0.10417	0.0999284	± 0.00489452	[0.0950339 , 0.104823]
0.25	0.10667	0.102409	± 0.00425326	[0.0981562 , 0.106663]
0.30	0.10989	0.107468	± 0.00471989	[0.102748 , 0.112188]
0.35	0.11396	0.114737	± 0.00515846	[0.109579 , 0.119896]
0.40	0.11905	0.112114	± 0.0051447	[0.106969 , 0.117258]
0.45	0.12539	0.122178	± 0.00528149	[0.116896 , 0.127459]
0.50	0.13333	0.127661	± 0.00551867	[0.122143 , 0.13318]
0.55	0.14337	0.140281	± 0.00604103	[0.13424 , 0.146322]
0.60	0.15625	0.157406	± 0.00639878	[0.151007 , 0.163805]
0.65	0.17316	0.1682424	± 0.00609569	[0.162146 , 0.174338]
0.70	0.19608	0.200341	± 0.00962453	[0.190717 , 0.209966]
0.75	0.22857	0.226234	± 0.0101855	[0.216049 , 0.23642]
0.80	0.27778	0.268359	± 0.010716	[0.257643 , 0.279075]
0.85	0.36036	0.346889	± 0.0140769	[0.332812 , 0.360966]
0.90	0.52632	0.490562	± 0.0231121	[0.46745 , 0.513674]
0.95	1.02564	0.978971	± 0.0416778	[0.937303 , 1.02066]

Cuadro 4.16: Sistema M/M/2: Tiempo estancia en el sistema (T) con nivel de confianza del 95 %

Siendo los tiempos de ejecución de la simulación y el número de iteraciones del algoritmo los siguientes:

ρ	**Tiempo simulado**	**Tiempo ejecución**	**Iteraciones**
0.05	~ 0	~ 0	2
0.10	~ 0	~ 0	2
0.15	~ 0	~ 0	3
0.20	~ 0	~ 0	2
0.25	~ 0	~ 0	2
0.30	~ 0	~ 0	3
0.35	~ 0	~ 0	5
0.40	~ 0	~ 0	1
0.45	~ 0	~ 0	5
0.50	~ 0	~ 0	6
0.55	~ 0	~ 0	8
0.60	~ 0	~ 0	8
0.65	18m 7s	0.367s	11
0.70	17m 5s	0.383s	11
0.75	32m 5s	0.722s	12
0.80	1h 15m	1.710s	14
0.85	3h 17m	4.771s	17
0.90	4h 38m	7.017s	18
0.95	18h	28.675s	22

Cuadro 4.17: Sistema M/M/2: Tiempo de ejecución e iteraciones con nivel de confianza del 95 %

La figura 4.13 muestra la comparación entre el valor teórico del tiempo en el sistema T y el valor medio obtenido en la simulación con el intervalo de confianza deseado:

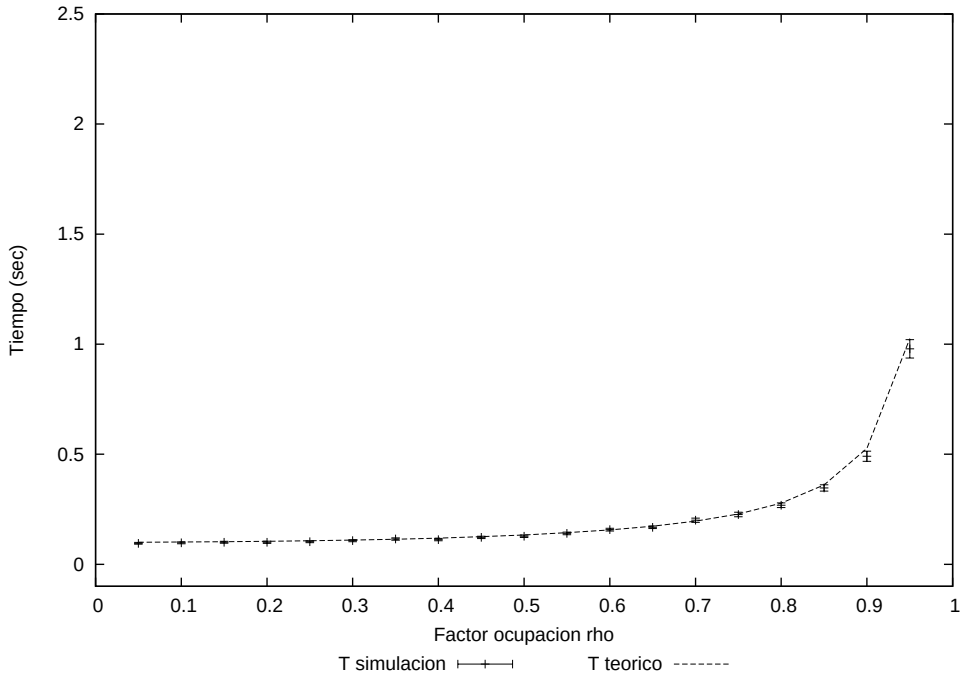

Figura 4.13: Sistema M/M/2: T teórico vs T simulado con nivel de confianza del 95 %

Por último, la figura 4.14 muestra la diferencia del número de iteraciones que realiza el algoritmo al variar el nivel de confianza exigido al método *batch means*.

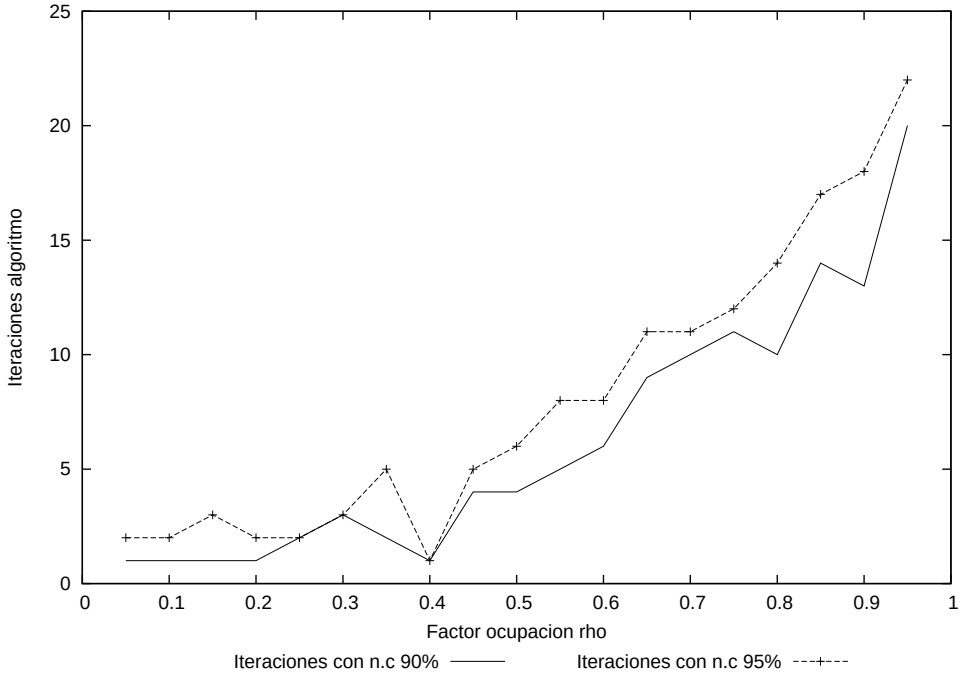

Figura 4.14: Sistema M/M/2: Comparativa iteraciones para el cálculo de T

Lógicamente a mayor nivel de confianza deseado, el mecanismo de *batch means* necesita más iteraciones en alcanzar un resultado satisfactorio.

4.3. Red de acceso compartido con protocolo Aloha

A continuación, se muestran los resultados obtenidos de simular un sistema formado por 20 hosts intentando comunicarse con un servidor central, utilizando un protocolo de acceso Aloha Puro y Aloha Ranurado, observando el número de colisiones que se producen en el canal y calculando la eficiencia media (S) de utilización del canal.

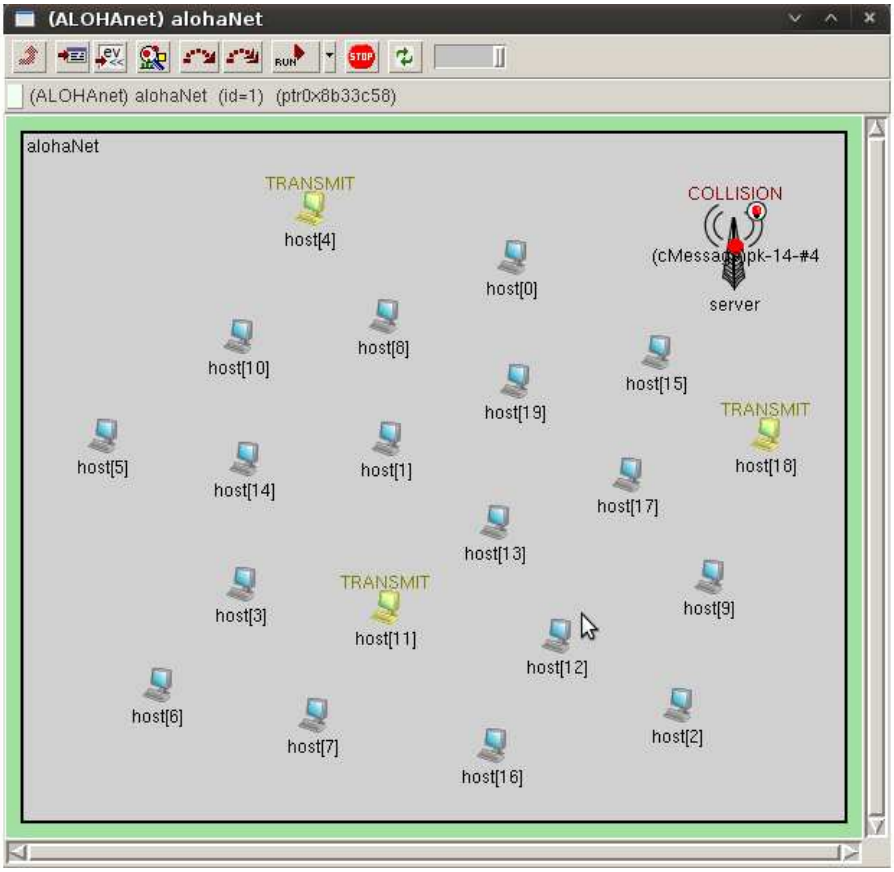

Figura 4.15: Vista en OMNeT++ de una red compartida con protocolo acceso Aloha

El tiempo de transmisión de los mensajes sigue una distribución exponencial de media $\mu = 0{,}1s$ y el tráfico generado se distribuye según una Poisson con probabilidad:

$$p(t) = \frac{(\lambda t)^k}{k!} \cdot e^{-\lambda t} \tag{4.5}$$

Siendo p(t) la probabilidad de k llegadas en un tiempo t, λ el tráfico generado. El tráfico cursado $S = G \cdot (1 - P_{colisión}) = G \cdot P_{no-colisión}$. En Aloha Simple, donde el intervalo vulnerable (donde se pueden producir colisiones) es $2t_t$

$$P_{no-colisión} = P(2t_t) = \frac{(\lambda t_t)^0}{0!} \cdot e^{\lambda 2t_t} \tag{4.6}$$

siendo t_t el tiempo de duración de una trama $G = \lambda t_t$, por lo que el tráfico cursado resulta:

$$S = G \cdot e^{-2G} \tag{4.7}$$

donde se asume que el proceso de llegada de tramas sigue una distribución de Poisson con un número medio de llegadas de 2G por cada $2t_t$ segundos. Un estudio más detallado de esta función nos hace ver que el máximo ocurre cuando $G = 0{,}5$, en el que $S = 0{,}18$. Obviamente esta función está muy lejos de lo esperado idealmente.

Los resultados obtenidos[1] después de simular este escenario son los siguientes:

G	S teórico	S simulado	Intervalo Confianza	Rango
0.1	0.0819	0.0813743	± 0.000368624	[0.0810057 , 0.0817429]
0.2	0.1341	0.134447	± 0.0000803836	[0.134367 , 0.134528]
0.4	0.1797	0.183901	± 0.00008644	[0.183814 , 0.183987]
0.5	0.1839	0.18796	± 0.000568742	[0.187391 , 0.188529]
0.8	0.1615	0.181589	± 0.00106762	[0.180521 , 0.182656]
1.0	0.1353	0.146622	± 0.000078724	[0.146549 , 0.146695]
1.5	0.0747	0.087313	± 0.000238204	[0.0870752 , 0.0875516]
2.0	0.0366	0.0454001	± 9.21286e-06	[0.0453909 , 0.0454093]
2.5	0.0168	0.021043	± 0.00122932	[0.0198137 , 0.0222723]
3.0	0.0074	0.0109573	± 0.000193836	[0.0107635 , 0.0111512]
3.5	0.0032	0.00594621	± 0.000570012	[0.0053762 , 0.00651622]
4.0	0.0013	0.002933	± 7.50603e-05	[0.00232446 , 0.00247458]

Cuadro 4.18: Aloha Puro: Utilización del canal (S) con nivel de confianza del 90 %

Como se observa, a medida que el tráfico ofrecido a la red aumenta, la eficiencia de utilización del canal S disminuye, se producen muchas más colisiones y el procedimiento obtiene peores resultados. Las muestras tienen valores mucho más correlados y el algoritmo finaliza antes pero los resultados no son tan óptimos. Cuando la red esta poco cargada, $0 < G < 0{,}5$ se obtienen valores más ajustados a los teóricos.

[1] Hay que recordar que el resultado teórico es una aproximación para un número infinito de estaciones que pueden transmitir. En nuestro caso, sólo se tienen 20, por lo que los resultados nunca podrán ser tan aproximados a los valores ideales

A continuación se muestran los tiempos de ejecución de la simulación, y las iteraciones realizadas por el método:

G	**Tiempo simulado**	**Tiempo ejecución**	**Iteraciones**
0.1	19ms	~ 0	7
0.2	2d 3h	2.1s	18
0.4	19h 11m	1.52s	17
0.5	96ms	~ 0	6
0.8	23ms	~ 0	5
1	2d 11h	10.19s	21
1.5	1h 49m	0.391s	11
2	12d 13h	73.869s	25
2.5	104ms	~ 0	1
3	38ms	~ 0	1
3.5	36m 39s	0.184s	5
4	16m 33s	0.134s	3

Cuadro 4.19: Aloha Puro: Tiempo de ejecución e iteraciones con nivel de confianza del 90 %

En la figura 4.16 se muestra la comparación entre el valor teórico de eficiencia del canal (S teórico) y el valor medio obtenido en la simulación con el intervalo de confianza deseado.

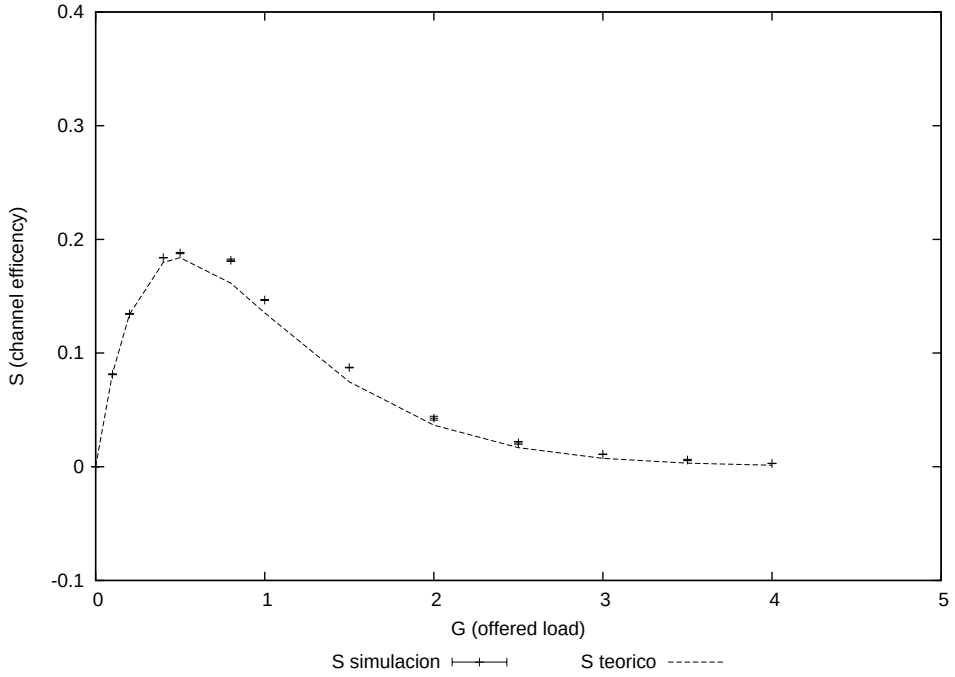

Figura 4.16: Aloha Puro: S teórico vs S simulado con nivel de confianza del 90 %

En el segundo caso, se utiliza el protocolo de Aloha ranurado. El procedimiento es el mismo, pero hay que tener en cuenta que el periodo vulnerable es la mitad, por lo que en este caso resulta:

$$S = G \cdot e^{-G} \tag{4.8}$$

donde se asume que el proceso de llegada de tramas sigue una distribución de Poisson con un número medio de llegadas de G por cada $2t_t$ segundos. Una estación no puede emitir en cualquier momento, sino justo al comienzo de una ranura, ocurriendo la máxima cuando $G = 1$, con un valor de $S = 0{,}37$, algo mayor que en el caso anterior (doble aproximadamente)

G	**S teórico**	**S simulado**	**Intervalo Confianza**	**Rango**
0.1	0.0905	0.0893763	± 0.000044572	[0.0893317 , 0.0894209]
0.2	0.1637	0.1581	± 0.00014261	[0.157958 , 0.158243]
0.4	0.2681	0.266865	± 0.000855751	[0.266009 , 0.267721]
0.5	0.3033	0.301437	± 0.000722151	[0.300715 , 0.302159]
0.8	0.3595	0.360281	± 0.000249966	[0.360031 , 0.360531]
1.0	0.3679	0.376194	± 0.000918212	[0.375276 , 0.377112]
1.5	0.3347	0.353563	± 0.000508042	[0.353055 , 0.354071]
2.0	0.2707	0.307248	± 0.000240984	[0.307007 , 0.307489]
2.5	0.2052	0.248599	± 0.000705309	[0.247894 , 0.249304]
3.0	0.1494	0.200936	± 0.00087393	[0.200062 , 0.20181]
3.5	0.1057	0.161881	± 0.000124063	[0.161757 , 0.162005]
4.0	0.0733	0.128	± 0.000355568	[0.127644 , 0.128355]

Cuadro 4.20: Aloha ranurado: Utilización del canal (S) con nivel de confianza del 90 %

Como era de esperar, la eficiencia de utilización del canal S aumenta, produciéndose menos colisiones, pero el procedimiento obtiene peores resultados. Las muestras tienen valores mucho más correlados y el algoritmo finaliza antes pero los resultados no son tan óptimos. Cuando la red esta poco cargada, $0 < G < 1$ se obtienen valores muy ajustados a los teóricos.

G	Tiempo simulado	Tiempo ejecución	Iteraciones
0.1	3d 13h	1.902s	18
0.2	3h 40m	0.160s	11
0.4	547ms	~ 0	10
0.5	1h 31m	0.173s	11
0.8	4h 57m	0.801s	15
1	48m 38s	0.191s	11
1.5	7h 30m	2.077s	17
2	5h 3m	1.795s	16
2.5	104ms	~ 0	8
3	122ms	~ 0	6
3.5	23h 43m	11.666s	21
4	32m 39s	0.296s	11

Cuadro 4.21: Aloha ranurado: Tiempo de ejecución e iteraciones con nivel de confianza del 90 %

A diferencia del caso anterior, el número de muestras recogidas por el método es mayor.

Por último, la figura 4.17 muestra una comparación entre el valor teórico con el valor obtenido en la simulación:

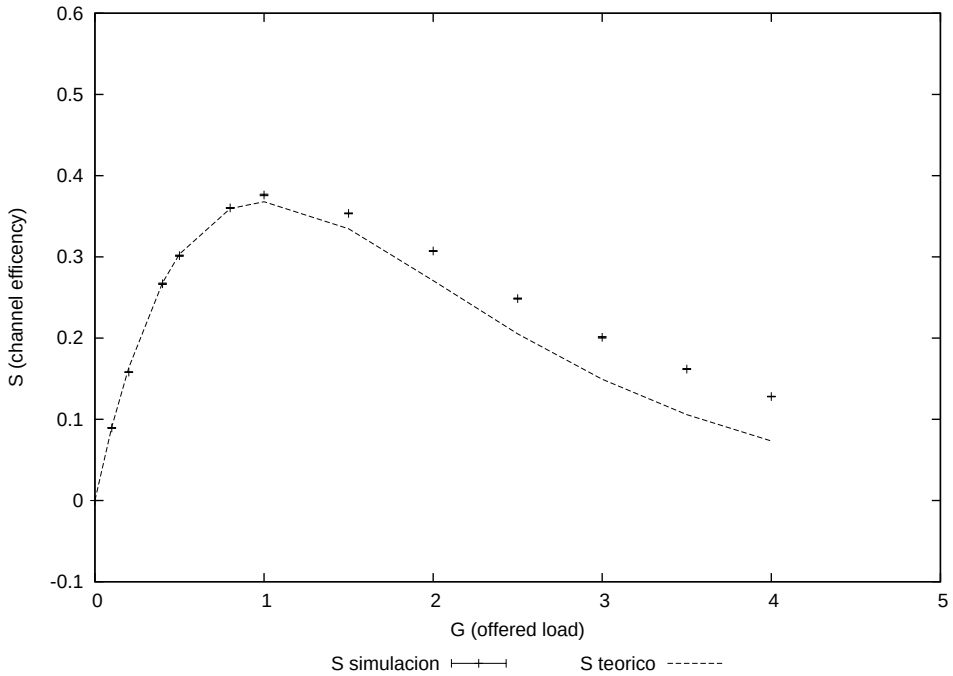

Figura 4.17: Aloha ranurado: S teórico vs S simulado con nivel de confianza del 90 %

En este último caso, los resultados son peores aunque siempre debemos tener en cuenta que para obtener los valores ideales deberíamos haber realizado la simulación con un número infinito de estaciones tratando de transmitir.

Capítulo 5

Historia del proyecto

En este capítulo se indica la duración total del proyecto así como el tiempo dedicado a cada una de sus partes. Así mismo, se mencionaron las decisiones tomadas y los replanteamientos llevados a cabo durante este tiempo.

- Documentación inicial: 120 horas.
  La primera parte del proyecto consistió en documentarse en dos campos concretos. El primero de ellos sobre el entorno de trabajo de la herramienta de simulación que vamos a utilizar. Paralelamente hubo que documentarse en los aspectos relativos a la teoría en que se basa el método de *batch means* para su posterior integración en el simulador OMNeT++.

- Herramienta de simulación OMNeT++: 200 horas. todos los conceptos se procedió a la fase de programación. Aquí se comenzó por la clase que implementara el algoritmo. Una vez hecho esto, procedimos a implementar un entorno de simulación. Para ello se decidió un escenario simple como es un sistema de colas. En el primer mes se desarrolló un modelo M/M/1 aprovechando las clases que ya estaban implementadas en OMNeT++. Sin embargo, la forma en que estaban implementadas estas clases no permitan aprovechar al máximo las posibilidades para obtener ciertas medidas estadísticas, y aprovechando nuevas características del simulador se desarrolló un sistema de colas M/M/k mucho más útil. En cambio si se utilizó el modelo para una red con protocolo de acceso Aloha, y únicamente se realizaron pequeños ajustes para incluir las librerías con el algoritmo de batch means.

- Pruebas: 120 horas.
  Una vez implementadas todas las clases, comprobamos los resultados del algoritmo. Principalmente nos centramos en estudiar los tiempos de espera en cola y en el sistema en el modelo M/M/k, y la eficiencia del tráfico cursado por el servidor en el escenario con el protocolo Aloha, analizando posteriormente si los resultados se aproximaban a los valores teóricos.

- Escritura de la memoria: 180 horas. Finalmente se escribió una memoria explicando el desarrollo del proyecto. Dicha memoria fue escrita con la herramienta LaTeX.

  Total Proyecto : 620 horas.

Capítulo 6

Conclusiones y trabajos futuros

Este capítulo ofrece una síntesis general del trabajo realizado y de posibles líneas de trabajo. Como hemos comprobado existen dos problemas principales en la interpretación de datos producidos por programas de simulación:

1. El estado transitorio en las simulaciones
2. La naturaleza correlada de los procesos de salida hace difícil la aplicación de técnicas habituales para el cálculo de intervalos de confianza

Este proyecto pretende reducir el tiempo de obtención de resultados de estimadores puntuales de régimen permanente en modelos de simulación por eventos discretos. Esto es, determinar un mecanismo fiable para acotar la duración de simulaciones sin horizonte finito. Sin embargo, se podría haber decidido abordar el problema con otro método distinto del *batch means*, por ejemplo el de las réplicas independientes. Una de las razones para la elección de este método se justificara en que la ejecución paralela permite romper las barreras que no lo hacen adecuado en un entorno secuencial, y que han fomentando el empleo de otras técnicas como en nuestro caso.

Law y Kelton (ver [4]), dan otras razones importantes para elegir este método. La aplicación paralela de la técnica de réplicas independientes tiene dos posibles vertientes: la aproximación basada en la ejecución de réplicas de tamaño fijo y la basada en la ejecución de una cantidad de réplicas fija a la que se les permite crecer.

El problema fundamental de las simulaciones es que en la mayoría de casos no producen observaciones que sean independientes e idénticamente distribuidas de manera normal. Como ejemplo, muchos autores comentan este problema aplicado al tiempo de espera de clientes consecutivos en un

complejo sistema de colas.

El método ideal para el análisis secuencial de la salida debería evitar en lo posible el sesgo producido por los efectos antes descritos, pero además no debería requerir una parametrización por parte del modelador, sino que, automáticamente, debería ser capaz de detectar y eliminar el transitorio inicial, y hacer el control de la longitud de la simulación hasta alcanzar el error relativo deseado a un determinado nivel de confianza.

La existencia de un método con borrado del transitorio produce estimaciones puntuales con menos sesgo e intervalos de confianza más estrechos y mejor cobertura de las simulaciones más cortas que sin borrado del transitorio inicial. Sin embargo, el método tiene problemas en la detección de algunos transitorios fuertes que se traducen en crecimientos lentos y que fuerzan al algoritmo a terminar prematuramente. Los buenos resultados obtenidos se enfrentan con la necesidad de unos profundos conocimientos de un analista de esta técnica y de su adecuada aplicación a cada tipo de modelo.

En cualquier caso, de los resultados se desprende que el tamaño ideal de muestra no es fijo y depende del modelo, y que la cobertura no está en función de la complejidad del modelo, pues un modelo sencillo como la cola M/M/1 ofrece peores resultados que el modelo complejo.

Un punto a tener en cuenta cuando se trabaja con simulaciones dirigidas mediante variables de entrada aleatorias es que las variables de salida resultantes también serán aleatorias. Por tanto, se deberán aplicar las técnicas estadísticas apropiadas a las variables resultantes si se desea analizarlas, interpretarlas y usarlas adecuadamente.

Como las simulaciones complejas suelen requerir grandes cantidades de tiempo de computación, los métodos estadísticos vistos en apartados anteriores suelen ser bastante costosos. Por tanto, un analista usar algún medio para incrementar la eficacia computacional, vista tanto como la adecuada codificación de la aplicación como de los adecuados métodos estadísticos.

Por ejemplo, el trabajo de Law y Kelton está orientado a la eficacia estadística desde el punto de vista de la varianza de las variables de salida. Si se puede de alguna forma reducir la varianza de una variable de salida aleatoria de interés sin alterar su esperanza, será posible obtener mayor precisión. Esto se puede traducir en intervalos de confianza más pequeños

para la misma cantidad de simulación o, alternativamente, conseguir la precisión deseada con menos esfuerzo computacional.

Los métodos para aplicar estas técnicas dependen normalmente del modelo particular a estudiar y, por tanto, requieren un conocimiento profundo del modelo de trabajo. Aún más, en general es imposible conocer de antemano la reducción de varianza a conseguir o, peor aún, si se reducir la varianza en comparación con los métodos directos. La aplicación de estas técnicas puede suponer un incremento del coste de desarrollo de la aplicación y del coste computacional, por lo que en ocasiones es interesante hacer pruebas piloto para comprobar si se obtienen ventajas.

En definitiva, las técnicas propuestas y su implementación práctica posibilitan el aprovechamiento de computadores heterogéneos para la realización masiva de experimentos de simulación en paralelo, permitiendo una notable reducción del tiempo de respuesta de los experimentos. Ajustándose a uno de los criterios de diseño propuestos, la aplicación de las técnicas es sencilla y extensible, en principio, a cualquier lenguaje de simulación por eventos discretos, con lo que el modelador puede aprovechar fácilmente las nuevas ventajas y olvidarse del problema del análisis estadístico de salida.

6.1. Líneas de investigación abiertas

Un factor primordial que puede limitar la aceleración de la simulación es el sesgo de las observaciones iniciales del método de las réplicas independientes, pues un incremento del número de réplicas aumenta la cantidad de muestras contaminadas que forman parte de la media, provocando una disminución de la cobertura. En este sentido se ha propuesto y desarrollado una técnica de precalentamiento basada en aprovechar el régimen permanente de una simulación previa para realizar la siguiente simulación, con lo que se espera que la transición al nuevo régimen permanente sea más rápida y menos sesgada. Como suele ser habitual ejecutar un modelo con distintos parámetros de entrada, esto se puede aprovechar para conseguir la propuesta anterior. En este sentido, se han desarrollado las técnicas que permiten al entorno paralelo reconfigurar un modelo en ejecución para que simule una configuración diferente de parámetros de entrada.

Se debe mejorar los aspectos relacionados con el análisis estadístico de los datos de salida de las réplicas. El análisis estadístico de los datos de salida debe ser la primera línea de trabajo a seguir, pues es la que garanti-

za, en última instancia, la calidad de los estimadores de las variables de salida.

La primera propuesta a investigar pasa por estudiar la posibilidad de que el mecanismo de instrumentación recoja información en forma de lotes, los cuales se calculan promediando una secuencia ordenada de observaciones. Así, por ejemplo, una réplica paralela puede dividir la secuencia de N observaciones en l lotes de tamaño m, y remitir al usuario la media de cada lote, con lo que la cantidad de información a transmitir se puede mantener arbitrariamente baja. Además, se cumple que la media de las observaciones generadas por la réplica es igual a la media de los promedios de los lotes, por lo que se siguen pudiendo aplicar directamente las propuestas hechas en este trabajo.

Parece aceptable que se pueda aumentar cualitativamente la potencia del método de análisis basado en réplicas independientes, optimizándose la eliminación del transitorio inicial. La idea de implementar un sistema de lotes facilita la construcción de otros analizadores estadísticos. Por ejemplo, sería inmediato desarrollar un analizador por lotes con la ventaja de que los lotes entre réplicas son independientes y, por tanto, de mayor calidad que en la versión secuencial.

Por otro lado, cuando se emplean máquinas con distinta potencia lo habitual es que unas réplicas sean más rápidas que otras, generando más observaciones. Una segunda línea de trabajo intentara mejorar el método de análisis estadístico para lograr estimadores de la esperanza y el intervalo de confianza en estos casos.

Apéndice A

Guía rápida de instalación de OMNeT++

Para más información leer la documentación disponible en la carpeta /doc del paquete de instalación.

A.1. Entorno Linux

1. Descargar el fichero fuente omnetpp-<version>-src.tgz[1]

2. Copiar el archivo omnetpp al directorio donde queramos instalarlo. Extraer el archivo usando el comando tar:

   ```
   tar zxvf omnetpp-<version>-src.tgz
   ```

   Se creará un subdirectorio llamado omnetpp-<version> que contendrá los ficheros de simulación. Deberá añadirse las siguientes líneas al fichero de inicio (.bashrc or .bash_profile si se está usando bash; .profile si se está usando algún otro sh como shell):

   ```
   export PATH=$PATH:~/omnetpp/bin
   export LD_LIBRARY_ PATH=$LD_LIBRARY_ PATH:~/omnetpp/lib
   ```

   Para que estas variables sean incluidas en el entorno, necesitará reiniciar la shell antes de seguir procediendo.

[1] http://omnetpp.org/omnetpp/doc_download/2061-omnet-40-source–ide–tgz

Asegurarse que los paquetes necesarios para compilar OMNeT++ están instalados en el sistema. En la distribución debian, por ejemplo, los siguientes paquetes son obligatorios:

```
sudo apt-get install build-essential gcc g++ bison flex perl tcl8.4
tcl8.4-dev tk8.4 tk8.4-dev blt blt-dev libxml2
libxml2-dev zlib1g zlib1g-dev libx11-dev
```

3. Comprobar el fichero configure.user para asegurarse de que contiene los ajustes y propiedades que necesitamos:

```
vi configure.user
```

y después ejecutar:

```
./configure
make
```

4. A partir de ahora podríamos probar todos los ejemplos y comprobar que funcionan correctamente. Por ejemplo, el ejemplo dyna arrancará una vez ejecutados los siguientes comandos:

```
cd /omnetpp/samples/dyna
./dyna
```

En la nueva versión, OMNeT++ 4.0, se incluye un nuevo entorno de desarrollo (basado en Eclipse). Por ello se necesita tener instalado en el sistema al menos una máquina virtual de Java (sun-java5-jre o sun-java6-jre u openjdk-6-jre deberían funcionar). Para lanzar el entorno simplemente ejecutamos:

```
omnetpp
```

Nota: Este entorno de desarrollo sólo es compatible en los siguientes sistemas:

- Windows 32bit
- Linux 32/64bit (i386)
- Mac OS X 10.4/5 (i386/ppc)

A.2. Entorno Windows

1. Descargar el fichero fuente omnetpp-<version>-src-windows.zip[2]

2. Copiar el archivo omnetpp al directorio donde queramos instalarlo (el nombre del directorio no debe contener espacios):

```
unzip omnetpp-<version>-src-windows.zip
```

   El archivo zip contiene una instalación de MSYS y MinGW así como todos los componentes adicionales (perl, tcl/tk, libxml2 etc). La única dependencia para OMNeT++ no incluida en el paquete es tener instalada una máquina virtual de Java, al menos la versión 1.5. Una vez creado el subdirectorio llamado omnetpp-<version>, ejecutamos:

```
mingwenv.cmd
```

   Y se abrirá una ventana de comandos desde donde podremos ejecutar y compilar nuestros proyectos. Si queremos hacerlo desde el explorador de windows, añadiremos el directorio a la variable PATH del sistema.

   En la nueva versión, OMNeT++ 4.0, se incluye un nuevo entorno de desarrollo (basado en Eclipse). Por ello se necesita tener instalado en el sistema al menos una máquina virtual de Java (Java Runtime Environment 5.0). Para ejecutar el entorno simplemente ejecutamos:

```
omnetpp
```

   Nota: Este entorno de desarrollo sólo es compatible en los siguientes sistemas:

   - Windows 32bit
   - Linux 32/64bit (i386)
   - Mac OS X 10.4/5 (i386/ppc)

   OMNeT++ ha sido testeado con el compilador MinGW gcc. La distribución actual contiene la versión de gcc 4.2.1. Microsoft Visual C++ no es soportada en la versión académica de OMNeT++.

[2]http://omnetpp.org/omnetpp/doc_download/2062-omnet-40-win32-source–ide–mingw-zip

Apéndice B

Guía rápida de ejecución de OMNeT++

B.1. Simulación con OMNeT++

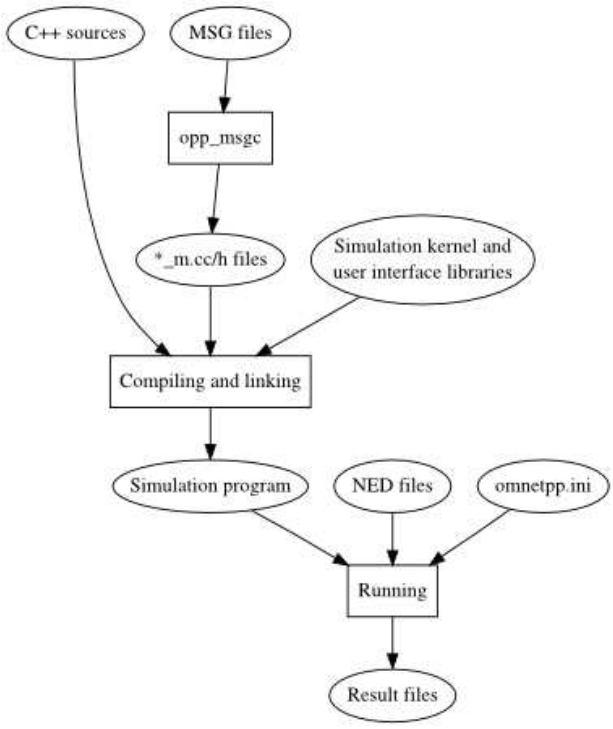

Figura B.1: Proceso de simulación en OMNeT++

El usuario debe compilar los ficheros fuente antes de lanzar la simulación

para que el intérprete busque posibles problemas de compatibilidad entre los diferentes módulos[1].

- Entorno Windows

```
opp_nmakemake -u <modo interfaz>-f -o <nombre_ejecutable>
<modo interfaz>: Cmdenv (modo consola) , Tkenv (modo gráfico)
nmake -f Makefile.vc
```

- Entorno Linux

```
opp_makemake -u <modo interfaz>-f -o <nombre_ejecutable>
<modo interfaz>: Cmdenv (modo consola) , Tkenv (modo gráfico)
make -f
```

Una vez realizado el anterior paso, OMNeT++ habrá generado algunos ficheros auxiliares para lanzar la aplicación y necesitaremos haber definido los parámetros de la simulación en un archivo de configuración[2]. Para lanzar la aplicación, bien en modo consola o bien en modo interfaz, lanzamos el ejecutable:

- Entorno Windows

```
nombre_ejecutable.exe -f fichero.ini
```

- Entorno Linux

```
./nombre_ejecutable -f fichero.ini
```

Si el fichero ini contiene más de una ejecución, esto es, varias ejecuciones que contienen distintos parámetros iniciales, podemos indicar al simulador que arranque directamente la ejecución deseada por medio de la opción:

- Versión 3.x

```
-r <número_ejecución>
```

- Versión 4.0

```
-c <nombre_configuración>
```

Si todo es correcto y hemos decidido usar el modo gráfico para realizar la simulación, deberá aparecer por pantalla lo siguiente:

[1]Para más información puede visitar:
http://omnetpp.org/doc/omnetpp40/ide-overview/ide-overview.html
http://www.omnetpp.org/doc/omnetpp33/manual/usman.html#sec288

[2]Por defecto el sistema busca el archivo *omnetpp.ini*

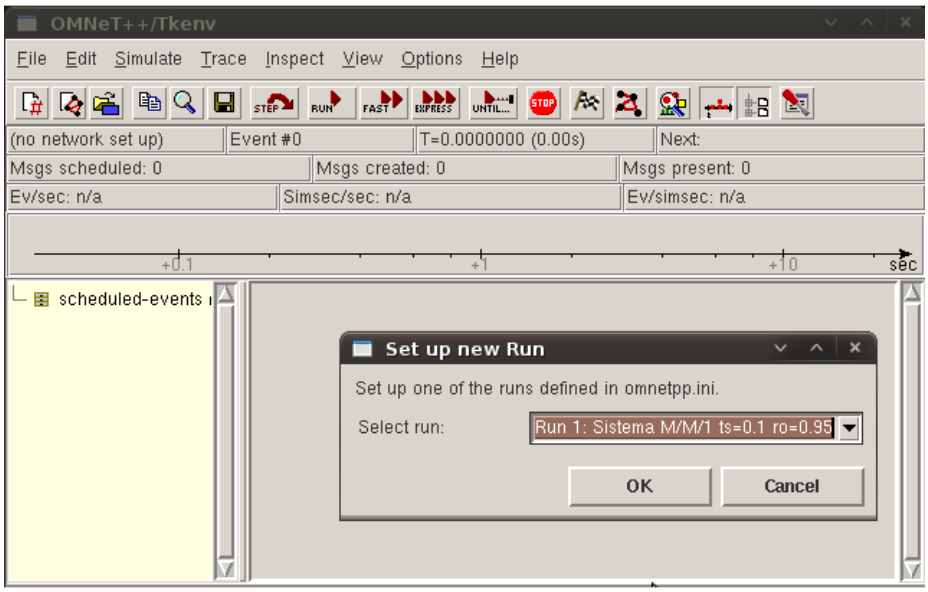

Figura B.2: Captura arranque de la simulación

Para comenzar la simulación, pulsamos el icono RUN o bien pulsamos F5. Si queremos aumentar la velocidad de la simulación podemos pulsar los botones de STEP (simulación evento a evento) RUN (velocidad normal) FAST (velocidad rápida) o de EXPRESS (velocidad muy rápida).

Para parar la simulación, pulsamos sobre el icono STOP o pulsando F8. Para finalizarla deberemos pulsar sobre el icono de FINISH (bandera a cuadros blancos y negros), y se mostrará información acerca de la simulación. Nótese que si se usa la librería *batchmeans* la simulación finalizará cuando el algoritmo termine de calcular una estimación de la media del parámetro que estamos evaluando.

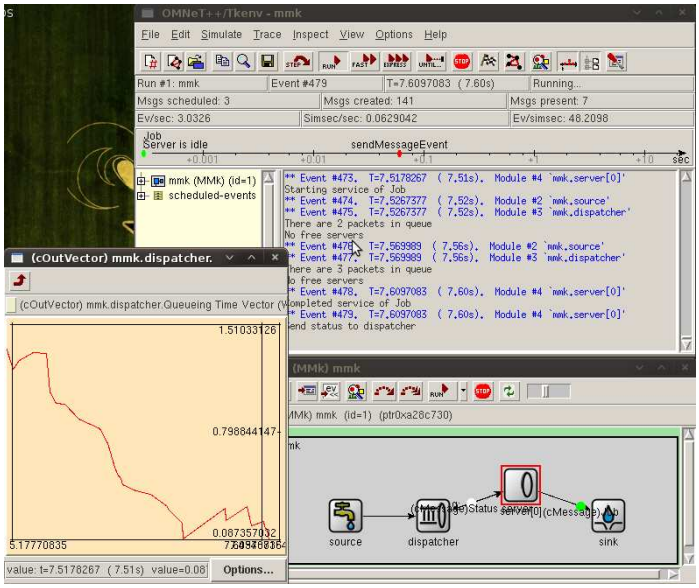

Figura B.3: Captura ejecución de la simulación

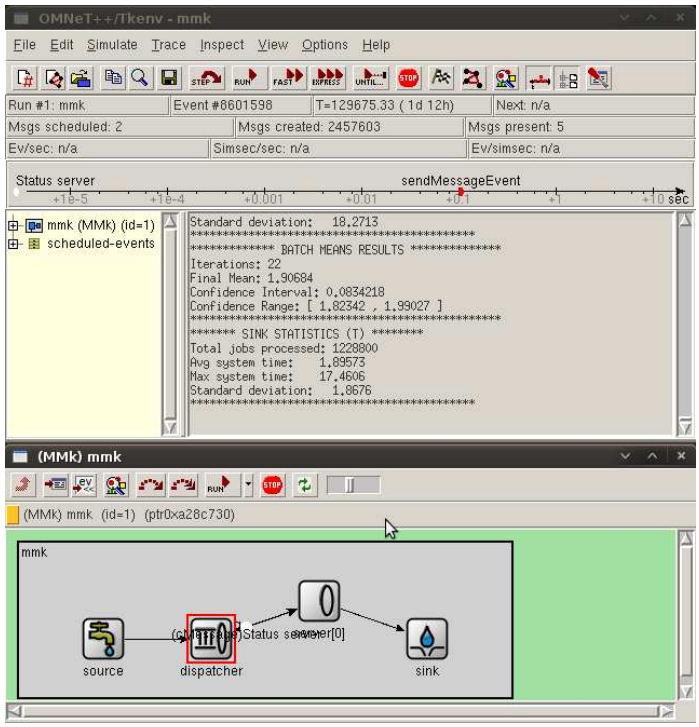

Figura B.4: Captura final de simulación

Apéndice C

Código Fuente Librería BatchMeans

```
//
// Copyright (C) 2008-2009 Iván de Gracia
// Implemented by Iván de Gracia <ivan.gracia@alumnos.uc3m.es>


#ifndef __CBATCHMEANS_H
#define __CBATCHMEANS_H

#define BatchesAccumulatorLength 400

#include <omnetpp.h>
#include <cdetect.h>

class SIM_API cADByBatchMeans : public cAccuracyDetection
{

    private:

        bool go;
        bool BatchEnded;
        double RelativePrecision, StudentPercentil;
        int IterationCounter;
        int CollectionCounter;

        double ConfidenceIntervalEstimate, FinalBatchMean;
        double BinaryBatchesAccumulator[BatchesAccumulatorLength];
        double TernaryBatchesAccumulator[BatchesAccumulatorLength];
```

```
    int BinaryAccumulatorIndex;
    int TernaryAccumulatorIndex;
    int AccumulatorIndex;

    double BinaryBatch;
    double TernaryBatch;

    double BinaryBatchSamples;
    double TernaryBatchSamples;

    int BinaryBatchIndex;
    int TernaryBatchIndex;
    int FinalBatchIndex;
    double CollectionSamples;

private:

    //Calculate estimator
    double JackKnifing(int InitialIndex, int EndIndex);
    //Calculate correlation
    double Correlation(int InitialIndex, int EndIndex);

public:

    /** @name Constructor, destructor, assignment. */

    /**
     * Default Constructor
     */

    explicit cADByBatchMeans(double ci, double cl);

    /**
     * Copy Constructor
     */

    cADByBatchMeans(const cADByBatchMeans& r);

     /**
      * Destructor
      */
```

```
virtual ~cADByBatchMeans();

/**
 * Assignment operator. The name member doesn't get copied;
 * see cObject's operator=() for more details.
 */

cADByBatchMeans& operator=(const cADByBatchMeans& res);

/**
 * Dupping is not implemented for this class. This function
 * gives an error (throws cRuntimeError) when called.
 */

virtual cPolymorphic *dup() const  {return new cADByBatchMeans(*this);}

/** @name Redefined cAccuracyDetection functions. */

/**
 * Collect a new sample for calculate mean
 */

virtual void collect(double Sample);

/**
 * Retuen true if we've got a desired confidence interval
 */

virtual bool detected() const;

virtual void stop() {go=false;}

virtual void start() {go=true;}

/**
 * Reset the algorithm to default values
 */

virtual void reset();

/** @name Setting up the detection object. */

void setParameters(double ci, double cl);
```

```
        /**
         * Print algorithm results
         */

        void PrintResults();
};

#endif

// Class cADByBatchMeans.
//
// Copyright (C) 2008-2009 Ivan de Gracia
// Implemented by Ivan de Gracia <ivan.gracia@alumnos.uc3m.es>


#define CorrelationThreshold 0.4
#define InitialSamples 600
#define StepSamples 200
#define FinalBatches 40

#include <cdetect.h>
#include "cbatchmeans.h"


cADByBatchMeans::cADByBatchMeans(double ci, double cl) : cAccuracyDetection()
{
    reset();
    setParameters(ci,cl);
}

cADByBatchMeans::cADByBatchMeans(const cADByBatchMeans& r) : cAccuracyDetection()
{
    setName(r.name());
    operator=(r);
}

cADByBatchMeans::~cADByBatchMeans()
{
}

cADByBatchMeans& cADByBatchMeans::operator=(const cADByBatchMeans& res)
{
```

```
    if (this==&res) return *this;

    cObject::operator=(res);

    go=res.go;
    BatchEnded=res.BatchEnded;
    RelativePrecision=res.RelativePrecision;
    StudentPercentil=res.StudentPercentil;
    CollectionCounter=res.CollectionCounter;
    IterationCounter=res.IterationCounter;
    BinaryAccumulatorIndex=res.BinaryAccumulatorIndex;
    TernaryAccumulatorIndex=res.TernaryAccumulatorIndex;
    BinaryBatchIndex=res.BinaryBatchIndex;
    TernaryBatchIndex=res.TernaryBatchIndex;
    CollectionSamples=res.CollectionSamples;
    BinaryBatchSamples=res.BinaryBatchSamples;
    TernaryBatchSamples=res.TernaryBatchSamples;
    BinaryBatch=res.BinaryBatch;
    TernaryBatch=res.TernaryBatch;
    pdf=res.pdf; pdfdata=res.pdfdata;
    return *this;

}

void cADByBatchMeans::reset()
{
    BatchEnded = false;
    CollectionCounter = 0;
    setParameters(0.1,0.9);

    for (AccumulatorIndex = 0 ; AccumulatorIndex < BatchesAccumulatorLength ;
         AccumulatorIndex++)
    {
        BinaryBatchesAccumulator[AccumulatorIndex] = 0.0;
        TernaryBatchesAccumulator[AccumulatorIndex] = 0.0;
    }

    BinaryAccumulatorIndex = 0;
    BinaryBatch = 0.0;
    BinaryBatchIndex = 0;
    TernaryAccumulatorIndex = 0;
    TernaryBatch = 0.0;
```

```
    TernaryBatchIndex = 0;
    IterationCounter = 1;
    CollectionSamples = InitialSamples+StepSamples;
    BinaryBatchSamples = 2;
    TernaryBatchSamples = 3;
}

void cADByBatchMeans::collect(double Sample)
{

    double FirstCorrelationEstimate,SecondCorrelationEstimate;
    double BatchVariance;
    int RelationIndex;
    double BatchesAccumulator[FinalBatches];
    int i;
    double AuxM,AuxV,x;
    int NItems;

    if(go)
    {
        CollectionCounter++;
        //AccumulateSample

        BinaryBatch = BinaryBatch+Sample;
        BinaryBatchIndex++;

        if (BinaryBatchIndex == BinaryBatchSamples)
        {
            BinaryBatchesAccumulator[BinaryAccumulatorIndex]
                        = BinaryBatch/BinaryBatchSamples;
            BinaryBatch = 0.0;
            BinaryBatchIndex = 0;
            BinaryAccumulatorIndex++;
        }

        TernaryBatch = TernaryBatch+Sample;
        TernaryBatchIndex++;



        if (TernaryBatchIndex == TernaryBatchSamples)
        {
            TernaryBatchesAccumulator[TernaryAccumulatorIndex]
```

```
          = TernaryBatch/TernaryBatchSamples;
    TernaryBatch = 0.0;
    TernaryBatchIndex = 0;
    TernaryAccumulatorIndex++;
}

// AccumulateSample

 if (CollectionCounter == CollectionSamples)
 {
    FirstCorrelationEstimate = JackKnifing(0,BatchesAccumulatorLength);
    //UpdateSampleAccumulator

    if ((IterationCounter % 2) != 0)
    {
        for (AccumulatorIndex = 0;
             AccumulatorIndex < (BatchesAccumulatorLength / 2);
             AccumulatorIndex++ )
        {
            BinaryBatchesAccumulator[AccumulatorIndex]
            = (BinaryBatchesAccumulator[AccumulatorIndex*2 + 1]
              + BinaryBatchesAccumulator[AccumulatorIndex*2])/2.0;
        }

        BinaryAccumulatorIndex = (BatchesAccumulatorLength / 2);
    }
    else
    {
        for (AccumulatorIndex = 0;
             AccumulatorIndex < (BatchesAccumulatorLength / 2);
             AccumulatorIndex++)
        {
            TernaryBatchesAccumulator[AccumulatorIndex]
            = (TernaryBatchesAccumulator[AccumulatorIndex*2 + 1]
              + TernaryBatchesAccumulator[AccumulatorIndex*2])/2.0;
        }

        TernaryAccumulatorIndex = (BatchesAccumulatorLength / 2);
    }

    SecondCorrelationEstimate = JackKnifing(0,BatchesAccumulatorLength / 2);

    if (FirstCorrelationEstimate < CorrelationThreshold)
```

```
{

    if ((FirstCorrelationEstimate <= 0.0)
         || (FirstCorrelationEstimate > SecondCorrelationEstimate))
    {

    //ConfidenceInterval

    RelationIndex = BatchesAccumulatorLength / FinalBatches;

    for (AccumulatorIndex = 0;
         AccumulatorIndex < FinalBatches;
         AccumulatorIndex++)
    {
         AuxM = 0.0;
         NItems = 0;

         for (i = AccumulatorIndex*RelationIndex;
              i < (AccumulatorIndex+1)*RelationIndex; i++)
         {
              if ((IterationCounter % 2) != 0)
                  AuxM = AuxM + BinaryBatchesAccumulator[i];
              else
                  AuxM = AuxM + TernaryBatchesAccumulator[i];
                  NItems++;
         }

         BatchesAccumulator[AccumulatorIndex] = AuxM/NItems;
    }

    AuxM = 0.0;
    AuxV = 0.0;
    NItems = 0;

    for (i = 0; i < FinalBatches; i++)
    {
         x = BatchesAccumulator[i];
         AuxM = AuxM + x;
         AuxV = AuxV + x*x;
         NItems++;
    }

    FinalBatchMean = AuxM/NItems;
```

```
            BatchVariance = AuxV/NItems - (FinalBatchMean*FinalBatchMean);

            ConfidenceIntervalEstimate = StudentPercentil
                                        *sqrt(BatchVariance/FinalBatches);

            if (ConfidenceIntervalEstimate < FinalBatchMean*RelativePrecision)
                 BatchEnded = true;
            else
            {
                 //IncrementIteration;
                 IterationCounter++;
                 CollectionSamples = CollectionSamples
                                    + StepSamples*pow(2.0,(IterationCounter/2));

                 BinaryBatchSamples = pow(2.0,((IterationCounter/2) + 1));

                 if ((IterationCounter % 2) == 0)
                     TernaryBatchSamples = 3*pow(2.0,((IterationCounter/2) - 1));
                 else
                     TernaryBatchSamples = 3*pow(2.0,(IterationCounter/2));
             }
        }
        else
        {

            //IncrementIteration
            IterationCounter++;
            CollectionSamples = CollectionSamples
                                + StepSamples*pow(2.0,(IterationCounter/2));
            BinaryBatchSamples = pow(2.0,((IterationCounter/2) + 1));

            if ((IterationCounter % 2) == 0)
                TernaryBatchSamples = 3*pow(2.0,((IterationCounter/2) - 1));
            else
                TernaryBatchSamples = 3*pow(2.0,(IterationCounter/2));
         }
}
else
{
    IterationCounter++;
    CollectionSamples = CollectionSamples
                            + StepSamples*pow(2.0,(IterationCounter/2));
       BinaryBatchSamples = pow(2.0,((IterationCounter/2) + 1));
```

```
          if ((IterationCounter % 2) == 0)
               TernaryBatchSamples = 3*pow(2.0,((IterationCounter/2) - 1));
          else
               TernaryBatchSamples = 3*pow(2.0,(IterationCounter/2));
     }
 }
}

void cADByBatchMeans::setParameters(double ci, double cl)
{

    RelativePrecision = ci;

    if ( cl <= 0.5 )
        StudentPercentil = 0.674;
    else if ( cl <= 0.6 )
        StudentPercentil = 0.842;
    else if ( cl <= 0.7 )
        StudentPercentil = 1.036;
    else if ( cl <= 0.8 )
        StudentPercentil = 1.282;
    else if ( cl <= 0.9 )
        StudentPercentil = 1.645;
    else if ( cl <= 0.95 )
        StudentPercentil = 1.960;
    else if ( cl <= 0.98 )
        StudentPercentil = 2.326;
    else if ( cl <= 0.99 )
        StudentPercentil = 2.576;
    else
        StudentPercentil = 2.878;
}


bool cADByBatchMeans::detected() const
{
    return BatchEnded;
}

void cADByBatchMeans::PrintResults()
{
```

```
    ev << "************* BATCH MEANS RESULTS **************" << endl;
    ev << "Iterations: " << IterationCounter << endl;
    ev << "Final Mean: " << FinalBatchMean << endl;
    ev << "Confidence Interval: " << ConfidenceIntervalEstimate << endl;
    ev << "Confidence Range: [ " << FinalBatchMean-ConfidenceIntervalEstimate
       << " , " << FinalBatchMean+ConfidenceIntervalEstimate << " ]" << endl;
    ev << "************************************************" << endl;
}

double cADByBatchMeans::JackKnifing(int InitialIndex, int EndIndex)
{

    double FstCorrelation, SndCorrelation, BatchesCorrelation;
    double JackKnifingEstimate;

    FstCorrelation = Correlation(InitialIndex,EndIndex / 2);
    SndCorrelation = Correlation((EndIndex / 2),EndIndex);
    BatchesCorrelation = Correlation(InitialIndex,EndIndex);
    JackKnifingEstimate = 2.0*BatchesCorrelation-(FstCorrelation+SndCorrelation)/2.0;

    return JackKnifingEstimate;
}

double cADByBatchMeans::Correlation(int InitialIndex, int EndIndex)
{

    int i;
    double CorrelationEstimate,MeanEstimate,Num,Den;
    double AuxM;
    int NItems;

    if ((IterationCounter % 2) != 0)
    {
        AuxM = 0.0;
        NItems = 0;

        for(i = InitialIndex; i < EndIndex-1; i++)
        {
            AuxM = AuxM + BinaryBatchesAccumulator[i];
            NItems++;
        }

        MeanEstimate = AuxM / NItems;
```

```
    Num = 0.0;
    Den = 0.0;

    for(i = InitialIndex; i < EndIndex;i++)
    {
        Num = Num + (BinaryBatchesAccumulator[i]
           - MeanEstimate)*(BinaryBatchesAccumulator[i+1] - MeanEstimate);
    Den = Den + (BinaryBatchesAccumulator[i]
           - MeanEstimate)*(BinaryBatchesAccumulator[i] - MeanEstimate);
}

Den = Den + (BinaryBatchesAccumulator[EndIndex] - MeanEstimate)
          * (BinaryBatchesAccumulator[EndIndex] - MeanEstimate);

if ( Den != 0.0 )
    CorrelationEstimate = Num/Den;
else
    CorrelationEstimate = 0.0;
 }
 else
 {
    AuxM = 0.0;
    NItems = 0;

    for(i = InitialIndex; i < EndIndex; i++)
    {
        AuxM = AuxM + TernaryBatchesAccumulator[i];
        NItems++;
    }

    MeanEstimate = AuxM / NItems;
    Num = 0.0;
    Den = 0.0;

    for(i = InitialIndex; i < EndIndex;i++)
    {
        Num = Num + (TernaryBatchesAccumulator[i] - MeanEstimate)
                  * (TernaryBatchesAccumulator[i+1] - MeanEstimate);
        Den = Den + (TernaryBatchesAccumulator[i] - MeanEstimate)
                  * (TernaryBatchesAccumulator[i] - MeanEstimate);
    }

    Den = Den + (TernaryBatchesAccumulator[EndIndex-1] - MeanEstimate)
```

```
                * (TernaryBatchesAccumulator[EndIndex] - MeanEstimate);

         if ( Den != 0.0 )
            CorrelationEstimate = Num/Den;
         else
            CorrelationEstimate = 0.0;
      }
      return CorrelationEstimate;
}
```

Bibliografía

[1] E. M. Fedriani Martel. *Guía rápida para el nuevo usuario de Latex*, July 2004.

[2] K. Holger. Leistungsbewertung and simulation: How to get data out of simulation programs. Technical report, 2006.

[3] A. M. Law and J. S. Carson. A sequential procedure for determining the length of a steady-state simulation. *Operations Research*, 27:1011–1025, 1979.

[4] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, third edition, 2000.

[5] M. Meketon and B. Schmeiser. Overlapping batch means: something for nothing. *Proceedings of the Winter Simulation Conference*, pages 227–230, 1984.

[6] D. Peña. *Fundamentos de Estadística*. Alianza Editorial, first edition, 2001.

[7] N. M. Steiger and J. R. Wilson. An improved batch means procedure for simulation output analysis. *Management Science, volume = 48, year = 2002, pages = 1569-1586*.

[8] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, fourth edition, 2002.

[9] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Wiley, second edition, 2001.

[10] N. van Forest. *Simulating Queueing with OMNeT++*, January 2003.

[11] A. Varga. *OMNeT++ API Reference*. http://www.omnetpp.org/doc/omnetpp33/api/index.html, 3.2 edition, October 2005.

[12] A. Varga. *OMNeT++ User Manual*. http://www.omnetpp.org/doc/omnetpp33/manual/usman.html, 3.2 edition, March 2005.

[13] A. Varga. *Migrating Omnet++ Simulations From Version 3.x To 4.0*. http://www.omnetpp.org/doc/omnetpp40/migration/migration.html, March 2009.

[14] A. Varga. *OMNeT++ API Reference*. http://www.omnetpp.org/doc/api/index.html, 4.0 edition, March 2009.

[15] A. Varga. *OMNeT++ User Manual*. http://www.omnetpp.org/doc/omnetpp40/manual/usman.html, 4.0 edition, March 2009.

[16] WikiBooks. *LaTeX*. http://en.wikibooks.org/wiki/LaTeX/, 2008.

[17] A. Willig. Steady-state simulations: Finding a steady-state mean value. Technical report.